Semi-Automatic Index Tuning: Keeping DBAs in the Loop

Karl Schnaitter
Aster Data
karl.schnaitter@asterdata.com

Neoklis Polyzotis
UC Santa Cruz
alkis@ucsc.edu

ABSTRACT

To obtain good system performance, a DBA must choose a set of indices that is appropriate for the workload. The system can aid in this challenging task by providing recommendations for the index configuration. We propose a new index recommendation technique, termed semi-automatic tuning, that keeps the DBA "in the loop" by generating recommendations that use feedback about the DBA's preferences. The technique also works online, which avoids the limitations of commercial tools that require the workload to be known in advance. The foundation of our approach is the Work Function Algorithm, which can solve a wide variety of online optimization problems with strong competitive guarantees. We present an experimental analysis that validates the benefits of semi-automatic tuning in a wide variety of conditions.

1 Introduction

Index tuning, i.e., selecting indices that are appropriate for the workload, is a crucial task for database administrators (DBAs). However, selecting the right indices is a very difficult optimization problem: there exists a very large number of candidate indices for a given schema, indices may benefit some parts of the workload and also incur maintenance overhead when the data is updated, and the benefit or update cost of an index may depend on the existence of other indices. Due to this complexity, an administrator often resorts to automated tools that can recommend possible index configurations after performing some type of workload analysis.

In this paper, we introduce a novel paradigm for index tuning tools that we term *semi-automatic index tuning*. A semi-automatic index tuning tool generates index recommendations by analyzing the workload online, i.e., in parallel with query processing, which allows the recommendations to adapt to shifts in the running workload. The DBA may request a recommendation at any time and is responsible for selecting the indices to create or drop. The most important and novel feature of semi-automatic tuning is that the DBA can provide feedback on the recommendation, which is taken into account for subsequent recommendations. In this fashion, the DBA can refine the automated recommendations by passing indirect domain knowledge to the tuning algorithm. Overall, the semi-automatic paradigm offers a unique combination of very desirable features: the tuner analyzes the running workload online and thus relieves the DBA from the difficult task of selecting a representative workload; the DBA retains total control over the performance-critical decisions to create or drop indices; and, the feedback mechanism couples human expertise with the computational power of an automated tuner to enable an iterative approach to index tuning.

We illustrate the main features of semi-automatic tuning with a simple example. Suppose that the semi-automatic tuner recommends to materialize three indices, denoted a, b, and c. The DBA may materialize a, knowing that it has negligible overhead for the current workload. We interpret this as implicit positive feedback for a. The DBA might also provide explicit negative feedback on c because past experience has shown that it interacts poorly with the locking subsystem. In addition, the DBA may provide positive feedback for another index d that can benefit the same queries as c without the performance problems. Based on this feedback, the tuning method can bias its recommendations in favor of indices a, d and against index c. For instance, a subsequent recommendation could be $\{a, d, e\}$, where e is an index that performs well with d. At the same time, the tuning method may eventually override the DBA's feedback and recommend dropping some of these indices if the workload provides evidence that they do not perform well.

Previous Work. Existing approaches to index selection fall in two paradigms, namely offline and online. Offline techniques [2, 6] generate a recommendation by analyzing a representative workload provided by the DBA, and let the DBA make the final selection of indices. However, the DBA is faced with the non-trivial task of selecting a good representative workload. This task becomes even more challenging in dynamic environments (e.g., ad-hoc data analytics) where workload patterns can evolve over time.

Online techniques [5, 10, 13, 14] monitor the workload and automatically create or drop indices. Online monitoring is essential to handle dynamic workloads, and there is less of a burden on the DBA since a representative workload is not required. On the other hand, the DBA is now completely out of the picture. DBAs are typically very careful with changes to a running system, so they are unlikely to favor completely automated methods.

None of the existing index tuning techniques achieves the same combination of features as semi-automatic tuning. Semi-automatic tuning starts with the best features from the two paradigms (online workload analysis with decisions delegated to the DBA) and augments them with a novel feedback mechanism that enables the DBA to interactively refine the recommendations. We note that interactive index tuning has been explored in the literature [7], but previous studies have focused on offline workload analysis. Our study is the first to propose an online feedback mechanism that is tightly coupled with the index recommendation engine.

.

A closer look at existing techniques also reveals that they cannot easily be modified to be semi-automatic. For instance, a naive approach to semi-automatic tuning would simply execute an online tuning algorithm in the background and generate recommendations based on the current state of the algorithm, but this approach ignores the fact that the DBA may select indices that contradict the recommendation. A key challenge of semi-automatic tuning is to adapt the recommendations in a flexible way that balances the influence of the workload and feedback from the DBA.

Our Contributions. We propose the WFIT index-tuning algorithm that realizes the new paradigm of semi-automatic tuning. WFIT uses a principled framework to generate recommendations that take the workload and user feedback into account. We can summarize the technical contributions of this paper as follows:

- We introduce the new paradigm of semi-automatic index tuning in Section 3. We identify the relevant design choices, provide a formal problem statement, and outline the requirements for an effective semi-automatic index advisor.

- We show that recommendations can be generated in a principled manner by an adaptation of the Work Function Algorithm [3] (WFA) from the study of metrical task systems (Section 4.1). We prove that WFA selects recommendations with a guaranteed bound on worst-case performance, which allows the DBA to put some faith in the recommended indices. The proof is interesting in the broader context of online optimization, since the index tuning problem does not satisfy the assumptions of the original Work Function Algorithm for metrical task systems.

- We develop the WFA$^+$ algorithm (Section 4.2) which uses a divide-and-conquer strategy with several instances of WFA on separate index sets. We show that WFA$^+$ leads to improved running time and better guarantees on recommendation quality, compared to analyzing all indices with a single instance of WFA. The guarantees of WFA$^+$ are significantly stronger compared to previous works for online database tuning [5, 10], and are thus of interest beyond the scope of semi-automatic index selection.

- We introduce the WFIT index-tuning algorithm that provides an end-to-end implementation of the semi-automatic paradigm (Section 5). The approach builds upon the framework of WFA$^+$, and couples it with two additional components: a principled feedback mechanism that is tightly integrated with the logic of WFA$^+$, and an online algorithm to extract candidate indices from the workload.

- We evaluate WFIT's empirical performance using a prototype implementation over IBM DB2 (Section 6). Our results with dynamic workloads demonstrate that WFIT generates online index recommendations of high quality, even when compared to the best indices that could be chosen with advance knowledge of the complete workload. We also show that WFIT can benefit from good feedback in order to improve further the quality of its recommendations, but is also able to recover gracefully from bad advice.

2 Preliminaries

General Concepts. We model the workload of a database as a stream of queries and updates Q. We let q_n denote the n-th statement and Q_N denote the prefix of length N.

Define $\mathcal{I}$ as the set of secondary indices that may be created on the database schema. The physical database design comprises a subset of $\mathcal{I}$ that may change over time. Given a statement $q \in Q$ and set of indices $X \subseteq \mathcal{I}$, we use $cost(q, X)$ to denote the cost of evaluating q assuming that X is the set of materialized indices. This function is possible to evaluate through the what-if interface of modern optimizers. Given disjoint sets $X, Y \subseteq \mathcal{I}$, we define $benefit_q(Y, X) = cost(q, X) - cost(q, Y \cup X)$ as the difference in query cost if Y is materialized in addition to X. Note that $benefit_q(Y, X)$ may be negative, if q is an update statement and Y contains indices that need to be updated as a consequence of q.

Another source of cost comes from adding and removing materialized indices. We let $\delta(X, Y)$ denote the cost to change the materialized set from X to Y. This comprises the cost to create the indices in $Y - X$ and to drop the indices in $X - Y$. The δ function satisfies the triangle inequality: $\delta(X, Y) \le \delta(X, Z) + \delta(Z, Y)$. However, δ is not a metric because indices are often far more expensive to create than to drop, and hence symmetry does not hold: $\delta(X, Y) \neq \delta(Y, X)$ for some X, Y.

Index Interactions. A key concern for index selection is the issue of *index interactions*. Two indices a and b interact if the benefit of a depends on the presence of b. As a typical example, a and b can interact if they are intersected in a physical plan, since the benefit of each index may be boosted by the other. Note, however, that indices can be used together in the same query plan without interacting. This scenario commonly occurs when indices are used to handle selection predicates on different tables.

We employ a formal model of index interactions that is based on our previous work on this topic [16]. Due to the complexity of index interactions, the model restricts its scope to some subset $\mathcal{J} \subseteq \mathcal{I}$ of interesting indices. (In our context, $\mathcal{J}$ is usually a set of indices that are relevant for the current workload.) The *degree of interaction* between a and b with respect to a query q is

$$doi_q(a, b) = \max_{X \subseteq \mathcal{J}} |benefit_q(\{a\}, X) - benefit_q(\{a\}, X \cup \{b\})|.$$

It is straightforward to verify the symmetry $doi_q(a, b) = doi_q(b, a)$ by expanding the expression of $benefit_q$ in the metric definition. Overall, this degree of interaction captures the amount that the benefits of a and b affect each other. Given a workload Q, we say a, b interact if $\exists q \in Q : doi_q(a, b) > 0$, and otherwise a, b are independent.

Let $\{P_1, \ldots, P_K\}$ denote a partition of indices in $\mathcal{J}$. Each P_k is referred to as a *part*. The partition is called *stable* if the cost function obeys the following identity for any $X \subseteq \mathcal{J}$:

$$cost(q, X) = cost(q, \emptyset) - \textstyle\sum_{k=1}^{K} benefit_q(X \cap P_k, \emptyset). \quad (2.1)$$

Essentially, a stable partition decomposes the benefit of a large set X into benefits of smaller sets $X \cap P_k$. The upshot for index tuning is that indices can be selected independently within each P_k, since indices from different parts have independent benefits. As shown in [16], the stable partition with the smallest parts is given by the connected components of the binary relation $\{(a, b) \mid a, b \text{ interact}\}$. The same study also provides an efficient algorithm to compute the binary relation and hence the minimum stable partition.

In the worst case, the connected components can be quite large if there are many complex index interactions. In practice, the parts can be made smaller by ignoring weak interactions, i.e., index-pairs (a, b) where $doi_q(a, b)$ is small. Equation (2.1) might not strictly hold in this case, but we can ensure that it provides a good approximation of the true query cost (that is still useful for index tuning) as long as the partition accounts for the most significant index interactions. We discuss this point in more detail in Section 5.

3 Semi-Automatic Index Tuning

At a high level, a semi-automatic tuning algorithm takes as input the current workload and feedback from the DBA, and computes a recommendation for the set of materialized indices. (Both inputs are continuous and revealed one "element" at a time.) The

DBA may inspect the recommendation at any time, and is solely responsible for scheduling changes to the materialized set. The online analysis allows the algorithm to adapt its recommendations to changes in the workload or in the DBA's preferences. Moreover, the feedback mechanism enables the DBA to pass to the algorithm domain knowledge that is difficult to obtain automatically. We develop formal definitions for these notions and for the overall problem statement in the following subsection.

We note that our focus is on the core problem of generating index recommendations, which forms the basic component of any index advisor tool. An index advisor typically includes other peripheral components, such as a user interface to visually inspect the current recommendation [9, 16] or methods to determine a materialization schedule for selected indices[16]. These components are mostly orthogonal to the index-recommendation component and hence we can reuse existing implementations. Developing components that are specialized for semi-automatic index tuning may be an interesting direction for future work.

3.1 Problem Formulation

Feedback Model. We use a simple and intuitive feedback model that allows the DBA to submit positive and negative votes according to current preferences. At a high level, a positive vote on index a implies that we should favor recommendations that contain a, until the workload provides sufficient evidence that a decreases performance. The converse interpretation is given for a negative vote on a. Our feedback model allows the DBA to cast several of these votes simultaneously. Formally speaking, the DBA expresses new preferences by providing two disjoint sets of indices $F^+, F^- \subseteq \mathcal{I}$, where indices in F^+ receive positive votes and indices in F^- receive negative votes.

We say that the DBA provides *explicit feedback* when they directly cast votes on indices. We also allow for *implicit feedback* that can be derived from the manual changes that the DBA makes to the index configuration. More concretely, we can infer a positive vote when an index is created and a negative vote when an index is dropped. The use of implicit feedback realizes an unobtrusive mechanism for automated tuning, where the tuning algorithm tailors its recommendations to the DBA's actions even if the DBA operates "out-of-band", i.e., without explicit communication with the tuning algorithm.

Problem Formulation. A semi-automatic tuning algorithm receives as input the workload stream Q and a stream V that represents the feedback provided by the DBA. Stream V has elements of the form $F = (F^+, F^-)$ per our feedback model. Its contents are not synchronized with Q, since the DBA can provide arbitrary feedback at any point in time. We only assume that Q and V are ordered in time, and we may refer to $Q \cup V$ as a totally ordered sequence. The output of the algorithm is a stream of recommended index sets $S \subseteq \mathcal{I}$, generated after each query or feedback element in $Q \cup V$. We focus on online algorithms, and hence the computation of S can use information solely from past queries and votes—the algorithm has absolutely no information about the future.

In order to complete the problem statement, we must tie the algorithm's output to the feedback in V. Intuitively, we consider the DBA to be an expert and hence the algorithm should trust the provided feedback. At the same time, the algorithm should be able to recover from feedback that is not useful for the subsequent statements in the workload. We bridge these somewhat conflicting goals by requiring each recommendation S to be *consistent* with recent feedback in V. To formally define consistency, let F_c^+ be the set of indices which have received a vote after the most recent query, where the most recent vote was positive. Define F_c^- analogously for negative votes. The consistency constraint requires S to contain all indices in F_c^+ and no indices in F_c^-, i.e., $F_c^+ \subseteq S \wedge S \cap F_c^- = \emptyset$.

Consistency forces recommendations to agree with the DBA's cumulative feedback so long as the algorithm has not analyzed a new query in the input. This property is aligned with the assumption that the DBA is a trusted expert. Moreover, consistency enables an intuitive interface in the case of implicit feedback that is derived from the DBA's actions: without the consistency constraint, it would be possible for the DBA to create an index a and immediately receive a recommendation to drop a (an inconsistent recommendation) even though the workload has not changed.

At the same time, our definition implies that $F_c^+ = F_c^- = \emptyset$ when a new query arrives. This says that votes can only force changes to the recommended configuration until the next query is processed, at which time the algorithm is given the *option* to override the DBA's previous feedback. Of course, the algorithm needs to analyze the workload carefully before taking this option, and determine whether the recent queries provide enough evidence to override past feedback. Otherwise, it could appear to the DBA that the system is ignoring the feedback and changing its recommendation without proper justification. Too many changes to the recommendation can also hurt the theoretical performance of an algorithm, as we describe later.

The Semi-Automatic Tuning Problem: *Given a workload Q and a feedback stream V of pairs (F^+, F^-), generate a recommended index set $S \subseteq \mathcal{I}$ after each element in $Q \cup V$ such that S obeys the consistency constraint.*

Note that user-specified storage constraints are not part of the problem statement. Although storage can be a concern in practice, the recommendation size is unconstrained because it is difficult to answer the question "How much disk space is enough?" before seeing the size of recommended indices. Instead, we allow the DBA to control disk usage when selecting indices from the recommendation.[1] To validate our choice, we conducted a small survey among DBAs of real-world installations. The DBAs were asked whether they would prefer to specify a space budget for materialized indices, or to hand-pick indices from a recommendation of arbitrary size. The answers were overwhelmingly in favor of the second option. One characteristic response said "Prefer hand-pick from DBA perspective, as storage is not so expensive as compared to overall objective of building a highly scalable system." This does not imply we should recommend all possible indices. On the contrary, as we see below, the recommendation must account for the overhead of materializing and maintaining the indices it recommends.

Performance Metrics. Intuitively, a good semi-automatic tuning algorithm should recommend indices that minimize the overall work done by the system, including the cost to process the workload as well as the cost to implement changes to the materialized indices. The first component is typical for index tuning problems and it reflects the quality of the recommendations. The second component stems from the online nature of the problem: the recommendations apply to the running state of the system, and it is clearly desirable to change the materialized set at a low cost. Low materialization cost is important even if new indices are built during a maintenance period, since these periods have limited duration and typically involve several other maintenance tasks (e.g., generation of usage reports, or backups).

Formally, let A be a semi-automatic tuning algorithm, and define

[1] Previous work [9, 16] and commercial systems provide tools to inspect index configurations, which may be adapted to our setting.

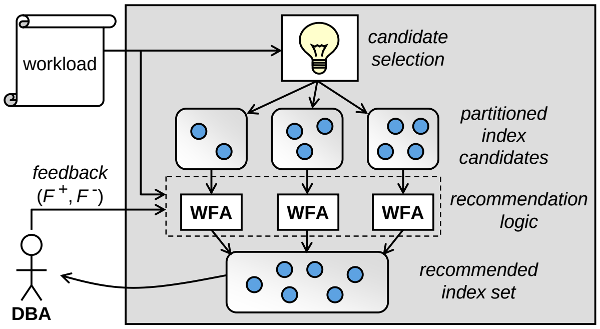


Figure 1: Components of the WFIT Algorithm.

S_n as the recommendation that A generates after analyzing q_n and all feedback up to q_{n+1}. Also denote the initial set of indices as S_0. We define the following *total work* metric that captures the performance of A's recommendations:

$$totWork(A, Q_N, V) = \sum_{1 \le n \le N} cost(q_n, S_n) + \delta(S_{n-1}, S_n)$$

The value of $totWork(A, Q_N, V)$ models the performance of a system where each recommendation S_n is adopted by the DBA for the processing of query q_n. This convention follows common practice in the field of online algorithms [3] and is convenient for the theoretical analysis that we present later. In addition, this model captures the effect of the feedback in V, as each S_n is required to be consistent (see above). Overall, total work forms an intuitive objective function, as it captures the primary sources of cost, while incorporating the effect of feedback on the choices of the algorithm. The adoption of this metric does not change the application of semi-automatic tuning in practice: the tuning algorithm will still generate a recommendation after each element in $Q \cup V$, and the DBA will be responsible for any changes to the materialized set.

It is clearly impossible for an online algorithm A to yield the optimal total work for all values of Q_N and V. Consequently, we adopt the common practice of *competitive analysis*: we measure the effectiveness of A by comparing it against an idealized *offline* algorithm OPT that has advance knowledge of Q_N and V and can thus generate optimal recommendations. Specifically, we say that A has *competitive ratio* c if $totWork(A, Q_N, V) \le c \cdot totWork(\text{OPT}, Q_N, V) + \alpha$ for any Q_N and V, where α is constant with respect to Q_N and V, and A and OPT choose recommendations from the same finite set of configurations. The competitive ratio c captures the performance of A compared to the optimal recommendations in the *worst case*, i.e., under some adversarial input Q_N and V. In this work, we assume that $V = \emptyset$ for the purpose of competitive analysis, since V comes from a trusted expert and hence the notion of adversarial feedback is unclear in practice. Our theoretical results demonstrate that the derivation of c remains non trivial even under this assumption. Applying competitive analysis to the general case of $V \neq \emptyset$ is a challenging problem that we leave for future work.

3.2 Overview of Our Solution

The remainder of the paper describes the WFIT algorithm for semi-automatic index tuning. Figure 1 illustrates WFIT's approach to generating recommendations based on the workload and DBA feedback. The approach starts with a *candidate selection* component, which generates indices that are relevant to the incoming queries. During candidate selection, WFIT also analyzes the interactions between candidate indices and uses these interactions to determine a stable partition of the candidates (see Section 2). Then the output of candidate selection is a partitioned set of indices, as shown in Figure 1. Once these candidates are chosen, WFIT analyzes the benefit of the indices with respect to the workload in order to generate the final recommendation. The logic that WFIT uses to generate recommendations is based on the Work Function Algorithm (WFA) of Borodin and El-Yaniv [3]. The original version of WFA was proposed for metrical task systems [4] but we extend its functionality to apply to semi-automatic index selection. A separate instance of WFA analyzes each part of the candidate set and only recommends indices within that part. As we discuss later, this divide-and-conquer approach of WFIT improves the algorithm's performance and theoretical guarantees. Finally, the DBA may request the current recommendation at any time and provide feedback to WFIT. The feedback is incorporated back into each instance of WFA and considered for the next recommendation.

The following two sections present the full details of each component of WFIT shown in Figure 1. Section 4 defines WFA and describes how WFIT leverages the array of WFA instances for its recommendation logic. Section 5 completes the picture, with the additional mechanisms that WFIT uses to generate candidates and account for DBA feedback.

4 A Work Function Algorithm for Index Tuning

The index tuning problem closely follows the study of task systems from online computation [4]. This allows us to base our recommendation algorithm on existing principled approaches. In particular, we apply the Work Function Algorithm [3] (WFA for short), which is a powerful approach to task systems with an optimal competitive ratio.

In order to fit the assumptions of WFA, we do not consider the effect of feedback and we fix a set of candidate indices $\mathcal{C} \subseteq \mathcal{I}$ from which all recommendations will be a drawn. In the next section, we will present the WFIT algorithm, which builds on WFA with support for feedback and automatic maintenance of candidate indices.

4.1 Applying the Work Function Algorithm

We introduce the approach of WFA with a conceptual tool that visualizes the index tuning problem in the form of a graph. The graph has a source vertex S_0 to represent the initial state of the system, as well as vertices (q_n, X) for each statement q_n and possible index configuration $X \subseteq \mathcal{C}$. The graph has an edge from S_0 to (q_1, X) for each X, and edges from (q_{n-1}, X) to (q_n, Y) for all X, Y and $1 < n \le N$. The weight of an edge is given by the transition cost between the corresponding index sets. The nodes (q, X) are also annotated with a weight of $cost(q, X)$. We call this the *index transition graph*. The key property of the graph is that the $totWork$ metric is equivalent to the sum of node and edge weights along the path that follows the recommendations. Figure 2 illustrates this calculation on a small sample graph. A previous study [2] has used this graph formulation for index tuning when the workload sequence is known a priori. Here, we are dealing with an online setting where the workload is observed one statement at a time.

The internal state of WFA records information about shortest paths in the index transition graph, where the possible index configurations comprise the subsets of the candidate set $\mathcal{C}$. More formally, after observing n workload statements, the internal state of WFA tracks a value denoted $w_n(S)$ for each index set $S \subseteq \mathcal{C}$, as defined in the following recurrence:

$$\begin{aligned} w_n(S) &= \min_{X \subseteq \mathcal{C}} \{w_{n-1}(X) + cost(q_n, X) + \delta(X, S)\} \quad (4.1) \\ w_0(S) &= \delta(S_0, S) \end{aligned}$$

We henceforth refer to $w_n(S)$ as the work function value for S after n statements. As mentioned above, the work function can be interpreted in terms of paths in the index transition graph. In the case where n is positive, $w_n(S)$ represents the sum of (i) the cost of the shortest path from S_0 to some graph node (q_n, X), and (ii) the transition cost from X to S. The actual value of $w_n(S)$ uses the $X \subseteq \mathcal{C}$ which minimizes this cost. We can think of $w_0(S)$ in a similar way, where the "path" is an empty path, starting and ending at S_0. Then the definition $w_0(S) = \delta(S_0, S)$ has a natural analogy to the recursive case.

Note that the total work of the theoretically optimal recommendations is equivalent to $totWork(Q_n, \text{OPT}, \emptyset) = \min_{S \subseteq \mathcal{C}}\{w_n(S)\}$. Hence, the intuition is that WFA can generate good recommendations online by maintaining information about the possible paths of optimal recommendations.

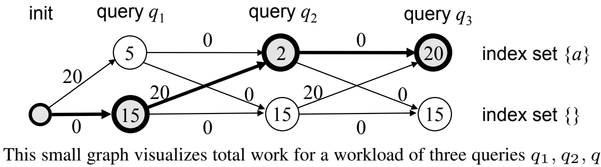
Figure 3 shows the pseudocode for applying WFA to index tuning. All of the bookkeeping in WFA is based on the fixed set $\mathcal{C}$ of candidate indices. The algorithm records an array $\mathbf{w}$ that is indexed by the possible configurations (subsets of $\mathcal{C}$). After analyzing the n-th statement of the workload, $\mathbf{w}[S]$ records the work function value $w_n(S)$. The internal state also includes a variable *currRec* to record the current recommendation of the algorithm.

The core of the algorithm is the *analyzeQuery* method. There are two stages to the method. The first stage updates the array $\mathbf{w}$ using the recurrence expression defined previously. The algorithm also creates an auxiliary array $\mathbf{p}$. Each $\mathbf{p}[S]$ contains index sets X such that a path from S_0 to (q_n, X) minimizes $w_n(S)$. The second stage computes the next recommendation to be stored in *currRec*. WFA assigns a numerical score to each configuration S as $score(S) = \mathbf{w}[S] + \delta(S, currRec)$ and the next state must minimize this score. To see the intuition of this criterion, consider a configuration X with a higher score than *currRec*, meaning that X cannot become the next recommendation. Then

$$\begin{aligned} & score(currRec) < score(X) \\ \Rightarrow \quad & w_n(currRec) - w_n(X) < \delta(X, currRec). \end{aligned}$$

The left-hand side of the final inequality can be viewed as the benefit of choosing a new recommendation X over *currRec* in terms of the total work function, whereas the right side represents the cost for WFA to "change its mind" and transition from X back to *currRec*. When the benefit is less than the transition cost, WFA will not choose X over the current recommendation. This cost-benefit analysis helps WFA make robust decisions (see Theorem 4.1).

The recommendation S chosen by WFA must also appear in $\mathbf{p}[S]$. Recall that $\mathbf{p}[S]$ records states X s.t. there exists a path from S_0 to (q_n, X) that minimizes $w_n(S)$. The condition specifies that $X = S$ for one such path, and hence $w_n(S) = w_{n-1}(S) + cost(q, S)$. An important result from Borodin et al. ([3], Lemma 9.2) shows that this condition is always satisfied by a state with minimum score. In other words, the criterion $S \in \mathbf{p}[S]$ is merely a tie-breaker for recommendations with the minimum score, to favor configurations whose work function does not include a transition after the last query is processed. This is crucial for the theoretical guarantees of WFA that we discuss later.



This small graph visualizes total work for a workload of three queries q_1, q_2, q_3, where recommendations are chosen between $\emptyset$ and $\{a\}$. The index a has cost 20 to create and cost 0 to drop. The highlighted path in the graph corresponds to an algorithm that recommends $\emptyset$ for q_1 and $\{a\}$ for q_2, q_3. The combined cost of edges and nodes in the path is $\delta(\emptyset, \emptyset) + cost(q_1, \emptyset) + \delta(\emptyset, \{a\}) + cost(q_2, \{a\}) + \delta(\{a\}, \{a\}) + cost(q_3, \{a\}) = 57$.

Figure 2: Index transition graph

```
Data: Set $\mathcal{C} \subseteq \mathcal{I}$ of candidate indices; Array $\mathbf{w}$ of work function values;
    Configuration $currRec$.
Initialization: Candidates $\mathcal{C}$ and initial state $S_0 \subseteq \mathcal{C}$ given as input;
    $\mathbf{w}[S] = \delta(S_0, S)$ for each $S \subseteq \mathcal{C}$; $currRec = S_0$.

Procedure WFA.$analyzeQuery(q)$
Input: The next statement $q$ in the workload
1 Initialize arrays $\mathbf{w}'$ and $\mathbf{p}$;
2 foreach $S \subseteq \mathcal{C}$ do
3     $\mathbf{w}'[S] = \min_{X \subseteq \mathcal{C}}\{\mathbf{w}[X] + cost(q, X) + \delta(X, S)\}$;
4     $\mathbf{p}[S] = \{X \subseteq \mathcal{C} \mid \mathbf{w}'[S] = \mathbf{w}[X] + cost(q, X) + \delta(X, S)\}$;
5 Copy $\mathbf{w}'$ to $\mathbf{w}$;
6 foreach $S \subseteq \mathcal{C}$ do $score(S) \leftarrow \mathbf{w}[S] + \delta(S, currRec)$;
7 $currRec \leftarrow \arg\min_{S \in \mathbf{p}[S]}\{score(S)\}$;

Function WFA.$recommend()$
1 return $currRec$;
```

Figure 3: Pseudocode for WFA.

EXAMPLE 4.1. *The basic approach of* WFA *can be illustrated using the scenario in Figure 2. The actual recommendations of* WFA *will be the same as the highlighted nodes. Before the first query is seen, the work function values are initialized as*

$$w_0(\emptyset) = 0, \quad w_0(\{a\}) = 20$$

based on the transition cost from the initial configuration $S_0 \equiv \emptyset$. After the first query, the work function is updated using (4.1)*:*

$$w_1(\emptyset) = 15, \quad w_1(\{a\}) = 25.$$

These values are based on the paths $\emptyset \rightarrow \emptyset$ and $\emptyset \rightarrow \{a\}$ respectively.[2] *The scores are the same as the respective work function values ($\delta(\emptyset, \emptyset) = \delta(\{a\}, \emptyset) = 0$ at line 6 of* WFA.*analyzeQuery), hence $\emptyset$ remains as* WFA*'s recommendation due to its lower score. After q_2, the work function values are both*

$$w_2(\emptyset) = w_2(\{a\}) = 27.$$

Both values use the path $\emptyset \rightarrow \{a\} \rightarrow \{a\}$. The calculation of $w_2(\emptyset)$ also includes the transition $\delta(\{a\}, \emptyset)$, which has zero cost. The corresponding scores are again equal to the work function, but here the tie-breaker comes into play: $\{a\}$ is preferred because it is used to evaluate q_2 in both paths, hence WFA *switches its recommendation to $\{a\}$. Finally, after q_3, the work function values are*

$$w_3(\emptyset) = 42, \quad w_3(\{a\}) = 47.$$

based on paths $\emptyset \rightarrow \{a\} \rightarrow \{a\} \rightarrow \emptyset$ and $\emptyset \rightarrow \{a\} \rightarrow \{a\} \rightarrow \{a\}$ respectively. The actual scores must also account for the current recommendation $\{a\}$. Following line 6 of WFA.*analyzeQuery,*

$$score(\emptyset) = 62, \quad score(\{a\}) = 47.$$

The recommendation of WFA *remains $\{a\}$, since it has a lower score. This last query illustrates an interesting property of* WFA:

[2] For example 4.1, we abuse notation and use index sets X in place of the graph nodes (q_n, X).

although the most recent query has favored dropping a, the recommendation does not change because the difference in work function values is too small to outweigh the cost to materialize a again. ∎

As a side note, observe that the computation of $w_n(S)$ requires computing $cost(q, X)$ for multiple configurations X. This is feasible using the what-if optimizer of the database system. Moreover, recent studies [12, 8] have proposed techniques to speed up successive what-if optimizations of a query. These techniques can readily be applied to make the computation of w_n very efficient.

WFA's Advantage: Competitive Analysis. WFA is a seemingly simple algorithm, but its key advantage is that we can prove strong guarantees on the performance of its recommendations.

Borodin and El-Yaniv [3] showed that WFA has a competitive ratio of $2\sigma - 1$ for any metrical task system with σ possible configurations, meaning that its worst-case performance can be bounded. Moreover, WFA is an optimal online algorithm, as this is the best competitive ratio that can be achieved. These are very powerful properties that we would like to transfer to the problem of index recommendations. However, the original analysis does not apply in our setting, since it requires δ to be a metric, and our definition of δ is not symmetric. One of the technical contributions of this paper is to show how to overcome the fact that δ is not a metric, and extend the analysis to the problem of index recommendations.

THEOREM 4.1. *The* WFA *algorithm, as shown in Figure 3, has a competitive ratio of* $2^{|\mathcal{C}|+1} - 1$. (Proof in the appendix)

This theoretical guarantee bolsters our use of WFA to generate recommendations. The competitive ratio ensures that the recommendations do not have an arbitrary effect on performance in the worst case. We show empirically in Section 6 that the average-case performance of the recommendations can be close to optimal. This behavior is appealing to DBAs, since they would not want to make changes that can have unpredictably bad performance.

4.2 Partitioning the Candidates

In the study of general task systems, the competitive ratio of WFA is theoretically optimal [4]. However, the algorithm has some drawbacks for the index recommendation problem, since it becomes infeasible to maintain statistics for every subset of candidates in $\mathcal{C}$ as the size of $\mathcal{C}$ increases. The competitive ratio $2^{|\mathcal{C}|+1} - 1$ also becomes nearly meaningless for moderately large sets $\mathcal{C}$. Motivated by these observations, we present an enhanced algorithm WFA$^+$, which exploits knowledge of index interactions to reduce the computational complexity of WFA, while enabling stronger theoretical guarantees.

The strategy of WFA$^+$ employs a stable partition $\{C_1, \ldots, C_K\}$ of $\mathcal{C}$, as defined in Section 2. The stable partition guarantees that indices in C_k do not interact with indices in any other part $C_l \neq C_k$. This is formalized by (2.1), which shows that each part C_i makes an independent contribution to the benefit. Moreover, it is straightforward to show that $\delta(X, Y) = \sum_k \delta(X \cap C_k, Y \cap C_k)$, i.e., we can localize the transition cost within each subset C_k. These observations allow WFA$^+$ to decompose the objective function $totWork$ into K components, one for each C_k, and then select indices within each subset using separate instances of WFA.

We define WFA$^+$ as follows. The algorithm is initialized with a stable partition $\{C_1, \ldots, C_K\}$ of $\mathcal{C}$, and initial configuration S_0. For $k = 1, \ldots, K$, WFA$^+$ maintains a separate instance of WFA, denoted WFA$^{(k)}$. We initialize WFA$^{(k)}$ with candidates C_k and initial configuration $S_0 \cap C_k$. The interface of WFA$^+$ follows WFA:

- WFA$^+$.$analyzeQuery(q)$ calls WFA$^{(k)}$.$analyzeQuery(q)$ for each $k = 1, \ldots, K$.
- WFA$^+$.$recommend()$ returns $\bigcup_k$ WFA$^{(k)}$.$recommend()$.

On the surface, WFA$^+$ is merely a wrapper around multiple instances of WFA, but the partitioned approach of WFA$^+$ provides several concrete advantages. The division of indices into a stable partition implies that WFA$^+$ must maintain statistics on only $\sum_k 2^{|C_k|}$ configurations, compared to the $2^{|\mathcal{C}|}$ states that would be required to monitor all the indices in WFA. This can simplify the bookkeeping massively: a back-of-the-envelope calculation shows that if WFA$^+$ is given 32 indices partitioned into subsets of size 4, then only 128 configurations need to be tracked, whereas WFA would require more than four billion states. We prove that this simplification is lossless, i.e., that WFA$^+$ selects the same indices as WFA.

THEOREM 4.2. *If* $\{C_1, \ldots, C_K\}$ *is a stable partition of* $\mathcal{C}$*, then* WFA$^+$ *on* $\{C_1, \ldots, C_K\}$ *will make the same recommendations as* WFA *on* $\mathcal{C}$. (Proof in the appendix)

It immediately follows that WFA$^+$ inherits the competitive ratio of WFA. However, the power of WFA$^+$ is that it enables a much smaller competitive ratio by taking advantage of the stable partition.

THEOREM 4.3. WFA$^+$ *has a competitive ratio of* $2^{c_{\max}+1} - 1$*, where* $c_{\max} = \max_k\{|C_k|\}$. (Proof in the appendix)

Hence the divide-and-conquer strategy of WFA$^+$ is a win-win, as it improves the computational complexity of WFA as well as the guarantees on performance. Observe that WFA$^+$ matches the competitive ratio of 3 that the online tuning algorithm of Bruno and Chaudhuri [5] achieves for the special case $|\mathcal{C}| = 1$ (the competitive analysis in [5] does not extend to a more general case). The competitive ratio is also superior to the ratio $\geq 8(2^{|\mathcal{C}|} - 1)$ for the OnlinePD algorithm of Malik et al. [10] for a related problem in online tuning.

5 The WFIT Algorithm

We introduced WFA$^+$ in the previous section, as a solution to the index recommendation problem with strong theoretical guarantees. The two limitations of WFA$^+$ are (i) it does not accept feedback, and (ii) it requires a fixed set of candidate indices and stable partition. In this section, we define the WFIT algorithm, which extends WFA$^+$ with mechanisms to incorporate feedback and automatically maintain the candidate indices.

Figure 4 shows the interface of WFIT in pseudocode. The methods $analyzeQuery$ and $recommend$ perform the same steps as the corresponding methods of WFA$^+$. In $analyzeQuery$, WFIT takes additional steps to maintain the stable partition $\{C_1, \ldots, C_K\}$. This work is handled by two auxiliary methods: $chooseCands$ determines what the next partition should be, and $repartition$ reorganizes the data structures of WFIT for the new partition. Finally, WFIT adds a new method $feedback$, which incorporates explicit or implicit feedback from the DBA.

In the next subsection, we discuss the $feedback$ method. We then provide the details of the $chooseCands$ and $repartition$ methods used by $analyzeQuery$.

5.1 Incorporating Feedback

As discussed in Section 3, the DBA provides feedback by casting positive votes for indices in some set F^+ and negative votes for a disjoint set F^-. The votes may be cast at any point in time, and the sets F^+, F^- may involve any index in $\mathcal{C}$ (even indices that are not part of the current recommendation). This mechanism is captured by a new method $feedback(F^+, F^-)$. The DBA can call $feedback$ explicitly to express preferences about the index configuration, and we also use *feedback* to account for the implicit feedback from manual changes to the index configuration.

Recall from Section 3 that the recommendations must be *consistent* with recent feedback, but should also be able to *recover* from poor feedback. Our approach to guaranteeing consistency is simple: Assuming that $currRec$ is the current recommendation, the new recommendation becomes $currRec - F^{-} \cup F^{+}$. Since WFIT forms its recommendation as $\bigcup_k currRec_k$, where $currRec_k$ is the recommendation from WFA running on part C_k, we need to modify each $currRec_k$ accordingly. Concretely, the new recommendation for C_k becomes $currRec_k - F^{-} \cup (F^{+} \cap C_k)$.

The recoverability property is trickier to implement properly. Our solution is to adjust the scores in order to appear as if the workload (rather than the feedback) had led WFIT to recommend creating F^{+} and dropping F^{-}. With this approach, WFIT can naturally recover from bad feedback if the future workload favors a different configuration. To enforce the property in a principled manner, we need to characterize the internal state of each instance of WFA after it generates a recommendation. Recall that WFA selects its next recommendation as the configuration that minimizes the *score* function. Let us assume that the selected configuration is Y, which differs from the previous configuration by adding indices Y^{+} and dropping indices Y^{-}. If we recompute *score* after Y becomes the current recommendation, then we can assert the following bound for each configuration S:

$$score(S) - score(Y) \geq \delta(S, S - Y^{-} \cup Y^{+}) + \delta(S - Y^{-} \cup Y^{+}, S) \qquad (5.1)$$

Essentially, this quantity represents the minimum threshold that $score(S)$ must overcome in order to replace the recommendation Y. Hence, in order for the internal state of WFA$^{(k)}$ to be consistent with switching to the new recommendation $currRec_k$, we must ensure that $score(S) - score(currRec_k)$, or the equivalent expression $\mathbf{w}^{(k)}[S] + \delta(S, currRec_k) - \mathbf{w}^{(k)}[currRec_k]$, respects (5.1). This can be achieved by increasing $\mathbf{w}^{(k)}[S]$ accordingly.

Figure 4 shows the pseudocode for *feedback* based on the previous discussion. For each part C_k of the stable partition, *feedback* first switches the current recommendation to be consistent with the feedback (line 4). Subsequently, it adjusts the value of $\mathbf{w}^{(k)}[S]$ for each $S \subseteq C_k$ to enforce the bound (5.1) on $score(S)$.

```
Data: Current set $\mathcal{C}$ of candidate indices;
    Stable partition $\{C_1, \ldots, C_K\}$ of $\mathcal{C}$;
    WFA instances WFA$^{(1)}, \ldots,$ WFA$^{(K)}$;
Initialization: Initial index set $S_0$ is provided as input;
    $\mathcal{C} = S_0$, $K = |S_0|$ and $C_i = \{a_i\}$ where $1 \leq i \leq |S_0|$ and
$a_1, \ldots, a_{|S_0|}$ are the indexes in $S_0$ ;
    for $k \leftarrow 1$ to $K$ do
        WFA$^{(k)} \leftarrow$ instance of WFA with candidates $C_k$
                    and initial configuration $C_k \cap S_0$

Procedure WFIT.$analyzeQuery(q)$
Input: The next statement $q$ in the workload.
1  $\{D_1, \ldots, D_M\} \leftarrow chooseCands(q)$ ;  // Figure 6
2  if $\{D_1, \ldots, D_M\} \neq \{C_1, \ldots, C_K\}$ then
       // Replace $\{C_1, \ldots, C_K\}$ with $\{D_1, \ldots, D_M\}$.
3      $repartition(\{D_1, \ldots, D_M\})$ ;  // Figure 5
4  for $k \leftarrow 1$ to $K$ do WFA$^{(k)}.analyzeQuery(q)$;

Function WFIT.$recommend()$
1  return $\bigcup_k$ WFA$^{(k)}.recommend()$;

Procedure WFIT.$feedback(F^{+}, F^{-})$
Input: Index sets $F^{+}, F^{-} \subseteq \mathcal{C}$ with positive/negative votes.
1  for $k \leftarrow 1$ to $K$ do
2      Let $\mathbf{w}^{(k)}$ denote the work function of WFA$^{(k)}$;
3      Let $currRec_k$ denote the current recommendation of WFA$^{(k)}$;
4      $currRec_k \leftarrow currRec_k - F^{-} \cup (F^{+} \cap C_k)$;
5      for $S \subseteq C_k$ do
6          $S^{\text{cons}} \leftarrow S - F^{-} \cup (F^{+} \cap C_k)$;
7          $minDiff \leftarrow \delta(S, S^{\text{cons}}) + \delta(S^{\text{cons}}, S)$;
8          $diff \leftarrow \mathbf{w}^{(k)}[S] + \delta(S, currRec_k) - \mathbf{w}^{(k)}[currRec_k]$;
9          if $diff < minDiff$ then
10             Increase $\mathbf{w}^{(k)}[S]$ by $minDiff - diff$;
```

Figure 4: Interface of WFIT.

5.2 Maintaining Candidates Automatically

The $analyzeQuery$ method of WFIT extends the approach of WFA^{+} to automatically change the stable partition as appropriate for the current workload. We present these extensions in the remainder of this section. We first discuss the $repartition$ method, which updates WFIT's internal state according to a new stable partition. Finally, we present $chooseCands$, which determines what that stable partition should be.

5.2.1 Handling Changes to the Partition

Suppose that the $repartition$ method is given a stable partition $\{D_1, \ldots, D_M\}$ for WFIT to adopt for the next queries. We require each of the indices materialized by WFA to appear in one of the sets $D_1, \ldots, D_M$, in order to avoid inconsistencies between the internal state of WFIT and the physical configuration. In this discussion, we do not make assumptions about how $\{D_1, \ldots, D_M\}$ is chosen. Later in this section, we describe how $chooseCands$ automatically chooses the stable partition that is given to $repartition$.

Unmodified Candidate Set. We initially consider the case where the new partition is over the same set of candidate indices, i.e., $\bigcup_{k=1}^{K} C_k = \bigcup_{m=1}^{M} D_m$. The original internal state of WFIT corresponds to a copy of WFA for each stable subset C_k. The new partition requires a new copy of WFA to be initialized for each new stable subset D_m. The challenge is to initialize the work function values corresponding to D_m in a meaningful way. We develop a general initialization method that maintains an equivalence between the work function values of $\{D_1, \ldots, D_M\}$ and $\{C_1, \ldots, C_K\}$, assuming that both partitions are stable.

We describe the reinitialization of the work function with an example. Assume the old stable partition is $C_1 = \{a\}, C_2 = \{b\}$, and the new stable partition has a single member $D_1 = \{a, b\}$. Let $\mathbf{w}^{(1)}, \mathbf{w}^{(2)}$ be the work function values maintained by WFIT for the subsets C_1, C_2. Let w_n be the work function that considers paths in the index transition graph with both indices a, b, which represents the information that would be maintained if a, b were in the same stable subset. In order to initialize work function values for D_1, we observe that the following identity follows from the assumption that $\{C_1, C_2\}$ is a stable partition:

$$w_n(S) = \mathbf{w}^{(1)}(S \cap \{a\}) + \mathbf{w}^{(2)}(S \cap \{b\}) - \sum_{1 \leq i \leq n} cost(q_i, \emptyset)$$

This is a special case of Lemma B.1, which we prove in Appendix B. The bottom line is that it is possible to reconstruct the values of the work function w_n using the work functions within the smaller partitions. For the purpose of initializing the state of WFA, the final sum may be ignored: the omission of this sum increases the scores of each state S by the same value, which does not affect the deci-

```
Procedure $\mathit{repartition}(\{D_1, \ldots, D_M\})$
Input: The new stable partition.
// Note: $D_1, \ldots, D_M$ must cover materialized indices
1 Let $\mathbf{w}^{(k)}$ denote the work function of WFA$^{(k)}$;
2 Let $\mathit{currRec}$ denote the current recommendation of WFIT;
3 for $m \leftarrow 1$ to $M$ do
4     Initialize array $\mathbf{x}^{(m)}$ and configuration variable $\mathit{newRec}_m$;
5     foreach $X \in 2^{D_m}$ do
6         $\mathbf{x}^{(m)}[X] \leftarrow \sum_{k=1}^{K} \mathbf{w}^{(k)}[C_k \cap X]$;
7         $\mathbf{x}^{(m)}[X] \leftarrow \mathbf{x}^{(m)}[X] + \delta(S_0 \cap D_m - \mathcal{C}, X - \mathcal{C})$;
8     $\mathit{newRec}_m \leftarrow D_m \cap \mathit{currRec}$;
9 Set $\{D_1, \ldots, D_M\}$ as the stable partition, where $D_m$ is tracked by a
  new instance WFA$^{(m)}$ with work function $\mathbf{x}^{(m)}$ and state $\mathit{newRec}_m$;
```

Figure 5: The *repartition* method of WFIT.

sions of WFA. Based on this reasoning, our repartitioning algorithm would initialize D_1 using the array $\mathbf{x}$ defined as follows:

$$\begin{aligned} \mathbf{x}[\emptyset] &\leftarrow \mathbf{w}^{(1)}[\emptyset] + \mathbf{w}^{(2)}[\emptyset] & \mathbf{x}[\{a\}] &\leftarrow \mathbf{w}^{(1)}[\{a\}] + \mathbf{w}^{(2)}[\emptyset] \\ \mathbf{x}[\{b\}] &\leftarrow \mathbf{w}^{(1)}[\emptyset] + \mathbf{w}^{(2)}[\{b\}] & \mathbf{x}[\{a,b\}] &\leftarrow \mathbf{w}^{(1)}[\{a\}] + \mathbf{w}^{(2)}[\{b\}] \end{aligned}$$

We use an analogous strategy to initialize the work function when repartitioning from D_1 to C_1, C_2:

$$\begin{aligned} \mathbf{w}^{(1)}[\emptyset] &\leftarrow \mathbf{x}[\emptyset] & \mathbf{w}^{(2)}[\emptyset] &\leftarrow \mathbf{x}[\emptyset] \\ \mathbf{w}^{(1)}[\{a\}] &\leftarrow \mathbf{x}[\{a\}] & \mathbf{w}^{(2)}[\{b\}] &\leftarrow \mathbf{x}[\{b\}] \end{aligned}$$

Again, note that these assignments result in work function values that would be different if C_1, C_2 were used as the stable partition for the entire workload. The crucial point is that each work function value is distorted by the same quantity (the omitted sum), so the difference between the scores of any two states is preserved.

The pseudocode for *repartition* is shown in Figure 5. For each new stable subset D_m, the goal is to initialize a copy of WFA with candidates D_m. The copy is associated with an array $\mathbf{x}^{(m)}$ that stores the work function values for the configurations in 2^{D_m}. For a state $X \subseteq D_m$, the value $\mathbf{x}^{(m)}[X]$ is initialized as the sum of $\mathbf{w}^{(k)}[X \cap C_k]$, i.e., the work function values of the configurations in the original partition that are maximal subsets of X (line 6). This initialization follows the intuition of the example that we described previously, since the stable partition $\{C_1, \ldots, C_K\}$ implies that $X \cap C_k$ is independent from $X \cap C_l$ for $k \neq l$. Line 7 makes a final adjustment for new indices in X, but this is irrelevant if the candidate set does not change (we will explain this step shortly). Finally, the current state corresponding to D_m is initialized by taking the intersection of $\mathit{currRec}$ with D_m.

Overall, *repartition* is designed in order for the updated internal state to select the same indices as the original state, provided that both partitions are stable. This property was illustrated in the example shown earlier. It is also an intuitive property, as two stable partitions record a subset of the same independencies, and hence both allow WFIT to track accurate benefits of different configurations. A more formal analysis of *repartition* would be worthwhile to explore in future work.

Modified Candidate Set. We now extend our discussion to the case where the new partition is over a different set of candidate indices, i.e., $\bigcup_{k=1}^{K} C_k \neq \bigcup_{m=1}^{M} D_m$. The *repartition* method (Figure 5) can handle this case without modifications. The only difference is that line 7 becomes relevant, and it may increase the work function value of certain configurations. It is instructive to consider the computation of $\mathbf{x}^{(m)}[X]$ when X contains an index a which did not previously appear in any C_k or the initial state S_0. Since a is a new index, it does not belong to any of the original subsets C_k, and hence the cost to materialize a will not be reflected in the sum $\sum_k \mathbf{w}^{(k)}[X \cap C_k]$. Since $\mathbf{x}^{(m)}[X]$ includes a transition to an index set with a materialized, we must add the cost to materialize a as a separate step. This idea is generalized by adding the transition cost on line 7. The expression is a bit complex, but we can explain it in an alternative form $\delta(S_0 \cap D_m - \mathcal{C}, X \cap D_m - \mathcal{C})$, which is equivalent because $X \subseteq D_m$. In this form, we can make an analogy to the initialization used for the work function before the first query, for which we use $w_0(X) = \delta(S_0, X)$. The expression used in line 7 computes the same quantity restricted to the indices $(D_m - \mathcal{C})$ that are new within D_m.

```
Data: Index set $\mathcal{U} \supseteq \mathcal{C}$ from which to choose candidate indices;
      Array $\mathit{idxStats}$ of benefit statistics for indices in $\mathcal{U}$;
      Array $\mathit{intStats}$ of interaction statistics for pairs of indices in $\mathcal{U}$.

Procedure $\mathit{chooseCands}(q)$
Input: The next statement $q$ in the workload.
Output: $D_1, \ldots, D_M$, a new partitioned set of candidate indices.
Knobs: Upper bound $\mathit{idxCnt}$ on number of indices in output;
       Upper bound $\mathit{stateCnt}$ on number of states $\sum_m 2^{|D_m|}$.
       Upper bound $\mathit{histSize}$ on number of queries to track in statistics
$\mathcal{U} \leftarrow \mathcal{U} \cup \mathit{extractIndices}(q)$;
$\mathit{IBG}_q \leftarrow \mathit{computeIBG}(q)$;  // Based on [16]
$\mathit{updateStats}(\mathit{IBG}_q)$;
$\mathcal{M} \leftarrow \{a \in \mathcal{C} \mid a \text{ is materialized}\}$;
$\mathcal{D} \leftarrow \mathcal{M} \cup \mathit{topIndices}(\mathcal{U} - \mathcal{M}, \mathit{idxCnt} - |\mathcal{M}|)$;
$\{D_1, \ldots, D_M\} \leftarrow \mathit{choosePartition}(\mathcal{D}, \mathit{stateCnt})$;
return $\{D_1, \ldots, D_M\}$;
```

Figure 6: The *chooseCands* Method of WFIT.

5.2.2 Choosing a New Partition

As the final piece of WFIT, we present the method *chooseCands*, which automatically decides the set of candidate indices $\mathcal{C}$ to be monitored by WFA, as well as the partition $\{C_1, \ldots, C_K\}$ of $\mathcal{C}$.

At a high level, our implementation of *chooseCands* analyzes the workload one statement at a time, identifying interesting indices and computing statistics on benefit interactions. These statistics are subsequently used to compute a new stable partition, which may reflect the addition or removal of candidate indices or changes in the interactions among indices. As we will see shortly, several of these steps rely on simple, yet intuitive heuristics that we have found to work well in practice. Certainly, other implementations of *chooseCands* are possible, and can be plugged in with the remaining components of WFIT.

The *chooseCands* method exposes three configuration variables that may be used to regulate its analysis. Variable *idxCnt* specifies an upper bound on the number of indices that are monitored by an instance of WFA, i.e., $\mathit{idxCnt} \geq |\mathcal{C}| = \sum_k |C_k|$. Variable *stateCnt* specifies an upper bound on the number of configurations tracked by WFIT, i.e., $\mathit{stateCnt} \geq \sum_k 2^{|C_k|}$. If the minimal stable partition does not satisfy these bounds, *chooseCands* will ignore some candidate indices or some interactions between indices, which in turn affects the accuracy of WFIT's internal statistics. Variable *histSize* controls the size of the statistics recorded for past queries. Any of these variables may be set to ∞ in order to make the statistics as exhaustive as possible, but this may result in high computational overhead. Overall, these variables allow a trade-off between the overhead of workload analysis and the effectiveness of the selected indices.

Figure 6 shows the pseudocode of *chooseCands*. The algorithm maintains a large set of indices $\mathcal{U}$, which grows as more queries are seen. The goal of *chooseCands* is to select a stable partition over some subset $\mathcal{D} \subseteq \mathcal{U}$. To help choose the stable partition, the algorithm also maintains statistics for $\mathcal{U}$ in two arrays: *idxStats* stores benefit information for individual indices and *intStats* stores information about interactions between pairs of indices within $\mathcal{U}$.

Given a new statement q in the workload, the algorithm first augments $\mathcal{U}$ with interesting indices identified by *extractIndices* (line 1). This function may be already provided by the database system (e.g., as with IBM DB2), or it can be implemented externally [1, 5]. Next, the algorithm computes the *index benefit graph* [16] (IBG for short) of the query (line 2). The IBG compactly encodes the costs of optimized query plans for all relevant subsets of $\mathcal{U}$. As we discuss later, *updateStats* uses the IBG to efficiently update the benefit and interaction statistics (line 3). The next step of the algorithm determines the new set of candidate indices $\mathcal{D}$ that should be monitored by WFIT for the upcoming workload, with an invocation of *topIndices* on line 5. We ensure that $\mathcal{D}$ includes the currently materialized indices (denoted $\mathcal{M}$), in order to avoid overriding the materializations chosen by WFA. Finally, *chooseCands* invokes *choosePartition* to determine the partition $D_1, \ldots, D_M$ of $\mathcal{D}$, and returns the result.

To complete the picture, we must describe the methodology that *topIndices* and *choosePartition* use to decide the new partition of indices, and the specific bookkeeping that *updateStats* does to enable this decision.

The $topIndices(X, u)$ **Method.** The job of $topIndices(X, u)$ is to choose at most u candidate indices from the set X that have the highest potential benefit.

We first describe the statistics used to evaluate the potential benefit of a candidate index. For each index a, the *idxStats* array stores entries of the form (n, β_n), where n is a position in the workload and β_n is the *maximum benefit* of a for query q_n. The maximum benefit is computed as $\beta_n = \max_{X \subseteq \mathcal{U}} benefit_{q_n}(\{a\}, X)$. The cell $idxStats[a]$ records the *histSize* most recent entries such that $\beta_n > 0$. These statistics are updated when *chooseCands* invokes *updateStats* on line 3. The function considers every index a that is relevant to q, and employs the IBG of query q in order to compute β_n efficiently. If $\beta_n > 0$ then (n, β_n) is appended to $idxStats[a]$ and the oldest entry is possibly expired in order to keep *histSize* entries in total.

Based on these statistics, $topIndices(X, u)$ returns a subset $Y \subseteq X$ with size at most u, which becomes the new set of indices monitored by WFIT. The first step of *topIndices* computes a "current benefit" for each index in X, which captures the benefit of the index for recent queries. We use $benefit^*_N(a)$ to denote the current benefit of a after observing N workload statements, and compute this value as follows. If $idxStats[a] = \emptyset$ after N statements, then $benefit^*_N(a)$ is zero. Otherwise, let $idxStats[a] = (n_1, b_1), \ldots, (n_L, b_L)$ such that $n_1 > \cdots > n_L$. Then

$$benefit^*_N(a) = \max_{1 \le \ell \le L} \frac{b_1 + \cdots + b_\ell}{N - n_\ell + 1}.$$

For each $\ell = 1, \ldots, L$, this expression computes an average benefit over the most recent $N - n_\ell + 1$ queries, and we take the maximum over all ℓ. Note that a large value of n_ℓ results in a small denominator, which gives an advantage to indices with recent benefit. This approach is inspired by the LRU-K replacement policy for disk buffering [11].

The second step of $topIndices(X, u)$ uses the current benefit to compute a score for each index in X, and returns the u indices with the highest scores. If $a \in X \cap \mathcal{C}$ (i.e., a is currently monitored by WFA), the score of a is simply $benefit^*(a)$. The score of other indices $b \in X - \mathcal{C}$ is $benefit^*(b)$ minus the cost to materialize b. This means that b requires extra evidence to evict an index in $\mathcal{C}$, which helps $\mathcal{C}$ be more stable.

The $choosePartition(\mathcal{D}, stateCnt)$ **method.** Conceptually, the stable partition models the strongest index interactions for recent queries. We first describe the statistics used to estimate the strength of interactions, and then the selection of the partition.

The statistics for *choosePartition* are based on the degree of interaction $doi_q(a, b)$ between indices $a, b \in \mathcal{U}$ for a workload statement q (Section 2). Specifically, we maintain an array *intStats* that is updated in the call to *updateStats* (which also updates *idxStats* as described earlier). The idea is to iterate over every pair (a, b) of indices in the IBG, and use the technique of [16] to compute $d \equiv doi_{q_n}(a, b)$. The pair (n, d) is added to $intStats[a, b]$ if $d > 0$, and only the *histSize* most recent pairs are retained.

We use $intStats[a, b]$ to compute a "current degree of interaction" for a, b after N observed workload statements, denoted as $doi^*_N(a, b)$, which is similar to the "current benefit" described earlier. If $intStats[a, b] = \emptyset$ then we set $doi^*_N(a, b) = 0$. Otherwise, let $intStats[a, b] = (n_1, d_1), \ldots, (n_L, d_L)$ for $n_1 > \cdots > n_L$, and

$$doi^*_N(a, b) = \max_{1 \le \ell \le L} \frac{d_1 + \cdots + d_\ell}{N - n_\ell + 1}.$$

To compute the stable partition, we conceptually build a graph where vertices correspond to indices and edges correspond to pairs of interacting indices. Then a stable partition is a clustering of the nodes so that no edges exist between clusters. In the context of *chooseCands*, we are interested in partitions $\{P_1, \ldots, P_M\}$ such that $\sum_m 2^{|P_m|} \le stateCnt$. Since there may exist no stable partition that obeys this bound, our approach is to ignore interactions until a feasible partition is possible. This corresponds to dropping edges from the conceptual graph, until the connected components yield a suitable clustering of the nodes.

An important question is which interactions to ignore. Our strategy is to minimize the error that the partition introduces in the formula for query cost (2.1), which is the basis of all statistics tracked by WFIT. It is straightforward to show that the error in (2.1) is bounded by the sum of *doi* values for ignored interactions. Hence we define the loss of a partition $P = \{P_1, \ldots, P_M\}$ as

$$loss(P) = \sum_{i<j} \sum_{a \in P_i} \sum_{b \in P_j} doi^*_N(a, b).$$

In the graph-based interpretation, this corresponds to the sum of edge weights for edges that cross clusters.

Figure 7 shows the pseudocode for function *choosePartition* that computes the new stable partition. The goal is to return a feasible partition that minimizes loss. We employ a randomized approach that finds several feasible partitions and returns the one with the least loss. As a baseline solution, the function considers the existing stable partition, augmented with singleton parts for the new indices in $\mathcal{D}$. It then performs RAND_CNT randomized iterations, where RAND_CNT is a parameter of the algorithm. Each iteration has two stages. The first stage simply merges singleton sets that exhibit a high degree of interaction. The pair $(\{a\}, \{b\})$ to merge is chosen randomly with weight proportional to $doi^*_N(a, b)$. The second stage is similar, with a different weighting scheme. Given two sets A and B, it assigns a weight proportional to $\sum_{a \in A} \sum_{b \in B} doi^*_N(a, b)/(2^{|A|+|B|} - 2^{|A|} - 2^{|B|})$. The normalization accounts for the additional number of configurations that will result from merging A and B. Essentially, the weight assigned to (A, B) represents the increase in $loss(P)$ per additional state tracked by WFIT, and hence the merging favors small

```
Function $choosePartition(\mathcal{D}, stateCnt)$
Input: Indices $\mathcal{D}$ to partition;
    Bound $stateCnt \geq \sum 2^{|P_m|}$ for the output $\{P_1, \ldots, P_M\}$
$bestSoln \leftarrow \emptyset;\ bestLoss \leftarrow \infty$;
// Try a baseline partition that is similar to the current one
Let $\{C_1, \ldots, C_K\}$ denote the current partition and $\mathcal{C} = \bigcup_k C_k$;
Initialize $\{C'_1, \ldots, C'_K\}$ by removing $\mathcal{C} - \mathcal{D}$ from $\{C_1, \ldots, C_K\}$;
$P \leftarrow \{C'_1, \ldots, C'_K\}$ ;
foreach $a \in \mathcal{D} - \mathcal{C}$ do Add $\{a\}$ to $P$;
if $P$ is feasible (i.e., satisfies the bound $stateCnt$) then
    $bestSoln \leftarrow P;\ bestLoss = loss(P)$ ;
// Try additional random partitions
for $i \leftarrow 1$ to RAND_CNT do
    $P \leftarrow$ a partition of $\mathcal{D}$ in singletons;
    while true do
        Let $\{P_1, \ldots, P_M\}$ denote the contents of $P$;
        $E \leftarrow \{\{P_i, P_j\} \mid loss(\{P_i, P_j\}) > 0 \wedge$ feasible to merge $P_i, P_j\}$;
        if $E = \emptyset$ then break;
        else if $E_1 \equiv \{\{P_i, P_j\} \in E \mid 1 = |P_i| = |P_j|\} \neq \emptyset$ then
            Choose random $\{P_i, P_j\} \in E_1$ with probability
            proportional to $loss(\{P_i, P_j\})$ ;
        else
            Choose $\{P_i, P_j\} \in E$ with probability proportional to
            $loss(\{P_i, P_j\})/(2^{|P_i|+|P_j|} - 2^{|P_i|} - 2^{|P_j|})$ ;
        $P \leftarrow$ result of merging $P_i, P_j$ in $P$ ;
    if $loss(P) < bestLoss$ then
        $bestSoln \leftarrow P$; $bestLoss \leftarrow loss(P)$ ;
return $bestSoln$;
```

Figure 7: Function *choosePartition***.**

sets whose indices have strong interactions. The function returns the best partition found across all iterations.

This concludes the final piece of the WFIT algorithm. As a final note, observe that the methods of WFIT use strategies that are quite orthogonal. In particular, the method *repartition* does not depend on the specific heuristics that *chooseCands* uses to determine the candidate indices. With this design, it is straightforward to substitute *chooseCands* with alternate strategies for candidate selection and partitioning. There is a broad design space for this component of WFIT, and this would be an interesting direction for future work.

6 Experimental Study

In this section, we present an empirical evaluation of WFIT using a prototype implementation that works as middleware on top of an existing DBMS. The prototype, written in Java, intercepts the SQL queries and analyzes them to generate index recommendations. The prototype requires two services from the DBMS: access to the what-if optimizer, and an implementation of the $extractIndices(q)$ method (line 1 in Figure 6). This design makes the prototype easily portable, as these services are common primitives found in index advisors [17, 1].

We conducted experiments using a port of the prototype to the IBM DB2 Express-C DBMS. The port uses DB2's design advisor [17] to provide what-if optimization and $extractIndices(q)$. Unless otherwise noted, we set the parameters of WFIT as follows: $idxCnt = 40$, $stateCnt = 500$, and $histSize = 100$. All experiments were run on a machine with two dual-core 2GHz Opteron processors and 8GB of RAM.

6.1 Methodology

Competitor Techniques. We compare WFIT empirically against two competitor algorithms. The first algorithm, termed BC, is an adaptation[3] of the state-of-the-art online tuning algorithm of Bruno and Chaudhuri [5]. BC analyzes the workload using ideas similar to WFIT, except that it always employs a stable partition corresponding to full index independence, i.e., each part contains a single index. After a query is analyzed, BC heuristically adjusts the measured index benefits to account for specific types of index interactions. The principled handling of index interactions is a major difference between WFIT and BC.

The second alternative is OPT, which has full knowledge of the workload and generates the optimal recommendations that minimize total work. OPT provides a baseline for the best-case performance of any online index recommendation algorithm.

In order to make a meaningful comparison between these algorithms, some of our experiments use a fixed set of candidates $\mathcal{C}$ and stable partition $\{C_1, \ldots, C_K\}$ throughout the workload. In this way, the algorithms select their recommendations from the same configuration space, and our experiments focus on the recommendation logic. This approach requires a simplification of WFIT so that *chooseCands* always returns $\{C_1, \ldots, C_K\}$. Our final experiment compares the simplified version of WFIT to the full version that allows *chooseCands* to modify the stable partition throughout the workload.

Data Sets and Workloads. We base the experimental study on an index tuning benchmark that we introduced in our previous work [15]. The benchmark is designed to stress test the effectiveness of online tuning algorithms, and it has already been used to compare existing methods. The benchmark simulates a system hosting multiple databases using the synthetic data sets TPC-C, TPC-H and TPC-E and the real-life data set NREF, with a total of 2.9GB of base-table data. We note that the database size is not a crucial statistic for our study, as we evaluate the performance of index-tuning algorithms using the optimizer's cost model (see discussion below).

We use the *complex* workload defined by the benchmark, which includes SQL query and update statements. Each statement involves a varying number of joins and selection predicates of mixed selectivity. The following is an example query from the workload:

```
SELECT count(*)
FROM tpce.security table1, tpce.company table2,
     tpce.daily_market table0
WHERE table1.s_pe BETWEEN 63.278 AND 86.091
  AND table1.s_exch_date BETWEEN '1995-05-12-01.46.40'
                             AND '2006-07-10-01.46.40'
  AND table2.co_open_date BETWEEN '1812-08-05-03.21.02'
                             AND '1812-12-12-03.21.02'
  AND table1.s_symb = table0.dm_s_symb
  AND table2.co_id=table1.s_co_id
```

And the following is an example update:

```
UPDATE tpch.lineitem
SET l_tax = l_tax + RANDOM_SIGN()*0.000001
WHERE l_extendedprice BETWEEN 65522.378 AND 66256.943
```

This update statement uses a user-defined function RANDOM_SIGN() which randomly returns 1 or -1 with equal probability.

The workload is separated in eight consecutive phases. Each phase comprises 200 statements and favors statements on specific data sets, thus requiring a different set of indices for effective tuning. Adjacent phases overlap in the focused data sets and also differ in the relative frequency of updates and queries. (See [15] for further details on data and SQL statements.) The specific workload is a difficult use case for index tuning due to the mix of updates and queries and the alternation of phases. In fact, the DB2 index

[3] The original algorithm was developed in the context of MS SQL Server. Some of its components do not have counterparts in DB2.

advisor was unable to recommend a beneficial index configuration for the whole workload, even with an infinite storage budget for indices. (We obtained similar experimental results with workloads of lower query complexity.)

Performance Metrics. We measure the performance of an online algorithm A using $totWork(A, Q_n, V)$ for the previously described workload and some feedback stream V. The definition of V depends on the experiment and is detailed when we present the results. As in previous studies on index tuning [5, 6, 15], the total work metric is evaluated using the optimizer's cost model. The goal is to isolate the performance of A from any cost-estimation errors, e.g., due to insufficient data statistics or faulty cost models.

In all experiments, the we measure the performance of A as $totWork(\text{OPT}, Q_n, V)/totWork(A, Q_n, V)$, which indicates the performance of A relative to the optimal recommendations of OPT. We note that the OPT can have very different recommendation schedules for Q_n and Q_{n+1} respectively, whereas A's recommendation schedule for Q_{n+1} is an extension of the schedule for Q_n.

We also report the overhead of algorithm A in terms of two components: the number of what-if optimization calls, and the remainder of the overhead as absolute wall-clock time. The reason for this separation is that the efficiency of the what-if optimizer is somewhat independent of the tuning algorithm. Indeed, techniques for very fast what-if optimization [8] can reduce substantially the overhead of any tuning task.

Generating the Fixed Stable Partition. As explained above, we choose a fixed stable partition $\{C_1, \ldots, C_K\}$ to be used by the competing algorithms. We developed an automated method to compute this partition in a way that captures the most relevant indices and interactions in the entire workload. Specifically, we first obtain a large set of interesting indices $\mathcal{U}$ by invoking DB2's index advisor on the read-only portion of the workload with an infinite space budget (as mentioned earlier, the index advisor would not recommend any indices to create for the entire workload). We then choose a subset $\mathcal{C} \subseteq \mathcal{U}$ and a partition of $\mathcal{C}$, using an offline variation of the $chooseCands$ algorithm. The only change to $chooseCands$ is to compute an average of the benefit and degree of interaction over the entire workload (rather than a suffix), and use these measurements as the criteria for the top indices and stable partition. For the workloads in our experiments, $\mathcal{U}$ contained roughly 300 indices, and the size of the stable partition depended on the parameter settings of WFIT.

6.2 Results

Baseline Performance. We begin with a baseline experiment where the stable partition is fixed and no feedback is provided ($V = \emptyset$). In this setting, WFIT becomes equivalent to WFA$^+$ (Section 4) and the measured performance reflects the effectiveness of the index recommendation logic. It also becomes possible to make a meaningful comparison to BC, which does not support feedback.

Figure 8 shows the normalized performance metrics for WFIT and BC. For WFIT we chart three curves that correspond to three different settings of 2000, 500, and 100 for the $stateCnt$ parameter of the stable partition. A high value corresponds to a more detailed stable partition that provides more information to WFIT but also increases its overhead. (The complexity of WFIT grows quadratically with $stateCnt$.) Figure 8 also includes a fourth curve labeled WFIT-IND, which corresponds to a variant of WFIT that considers all indices to be independent. In other words, this version of the algorithm assumes $doi_q(a, b) = 0$ for all indices and queries, which means that each index is in a separate singleton part. This version of WFIT would not be used in practice, but we show its performance in order to see the value of analyzing index interactions.

As shown, the quality of recommendations degrades gracefully as $stateCnt$ decreases from 2000 down to 100, with the overall difference remaining small throughout. The drop in performance is more significant for WFIT-IND, where all index interactions are ignored. We performed experiments with higher settings of $stateCnt$, up to 10000, but we omit the results, as there was very little difference compared to $stateCnt = 2000$. Essentially, the results show that WFIT can generate effective recommendations as long as the stable partition captures the important interactions among the candidate indices.

Another observation from Figure 8 is that WFIT's performance comes very close to the algorithm that has complete knowledge of the workload. The difference is less than 10% at the end, which is very significant if one considers the complex mix of updates and join queries in the workload. It is interesting to examine this empirical performance against the theoretical competitive ratio stated in Section 4. For this particular experiment, there are 8 indices in the biggest part of the stable partition and hence the performance of WFIT should always be within a factor of $2^{8+1} - 1$ of optimal. As shown by the results, WFIT's performance can be much better compared to this worst-case bound.

Finally, Figure 8 shows that WFIT outperforms BC by a significant margin. The difference becomes substantial after the initial statements in the workload, and by the end WFIT (without the independence assumption) attains >90% of the performance of OPT compared to 65% for BC. The difference shows that WFIT's principled handling of index interactions is more effective than the heuristics used by BC. In fact, the results show that even WFIT-IND outperforms BC on this workload. This could be due in part to the fact that our adaptation of BC is implemented outside the DBMS, and the original design of BC may be better suited for an internal implementation that is closely integrated with the query optimizer.

Overhead. For the same experiment, the Java implementation of WFIT on top of DB2 required 300ms on average to analyze each query and generate the recommendations. This magnitude of overhead is acceptable if one considers the much higher query execution cost and the savings obtained from having the right indices materialized. Still, overhead can be reduced substantially with a careful implementation inside the DBMS, or by switching to a lower value for $stateCnt$. For instance, setting $stateCnt = 100$ will not affect significantly the quality of recommendations (see Figure 8) but it can reduce the overhead by a factor of 25. A different solution is to do the analysis in a separate machine (e.g., the DBA's workstation) without any impact on normal query evaluation.

Regarding the number of what-if optimizations, WFIT averaged between 5 and 100 calls per query close to the start and end of the experiment respectively. The number of what-if calls is directly correlated with the number of candidate indices that are mined from the workload. A different implementation of WFIT could constrain the latter, but the experimental results of Bruno and Nehme [8] suggest that it is possible to perform 100 what-if calls per query while keeping up with the flow of the workload.

The Effect of Feedback. The next set of experiments evaluates WFIT's feedback mechanism (Section 5.1), one of the core features of the semi-automatic tuning paradigm.

We examine the performance of WFIT for two contrasting models of DBA feedback. The first model, represented with a feedback input V_{GOOD}, represents "good" feedback where the DBA casts a positive (resp. negative) vote for index a at point n in the workload if OPT creates (resp. drops) a after analyzing query n. The idea is

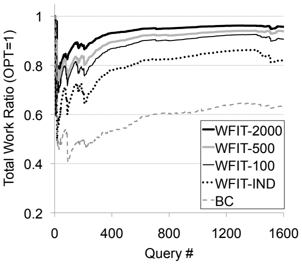


Figure 8: Baseline performance evaluation.

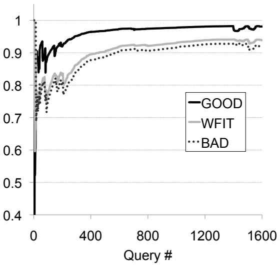


Figure 9: Effect of DBA's feedback.

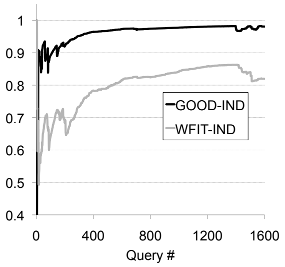


Figure 10: Effect of DBA's feedback under independence assumption

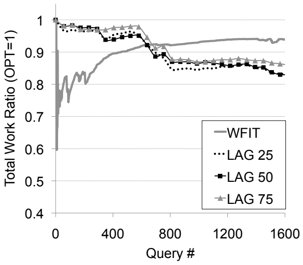


Figure 11: Effect of delayed responses.

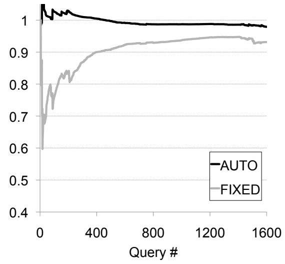


Figure 12: Automatic maintenance of stable partition.

to model a prescient DBA who can use votes to guide WFIT toward the optimal design. We also create a "bad" feedback input, denoted as V_{BAD}, as the mirror image of good feedback, i.e., we replace the positive votes with negative votes and vice versa.

Figure 9 shows the performance of WFIT for $V = V_{\text{GOOD}}$ and $V = V_{\text{BAD}}$. As a baseline, we include a run of WFIT without feedback, i.e., $V = \emptyset$. The results show that the feedback mechanism works intuitively. The useful feedback improves the performance of the baseline and pushes it closer to the optimal algorithm. WFIT does not exactly match the performance of OPT, since the latter computes its recommendations using much more detailed information (recall that WFIT uses a fixed stable partition with $stateCnt = 500$). The bad feedback causes a degradation of performance, as expected, but WFIT is still able to output effective recommendations and remain above 90% of optimal by the end of the workload. The key point is that WFIT initially biases its recommendations according to the erroneous feedback, but it is able to recover based on the subsequent analysis of the workload.

It is also interesting to examine the effect of feedback in the modified version of WFIT which assumes all indices are independent. This experiment models an interesting scenario for the usefulness of semi-automatic tuning, as the assumption of index independence can introduce significant errors in WFIT's internal statistics on index benefits, and hence DBA feedback can have a significant effect on the quality of the generated recommendations. Figure 10 shows the result of providing feedback as V_{GOOD} for the WFIT-IND algorithm. (We omit results that combine WFIT-IND with the "adversarial" feedback V_{BAD}, since such a scenario would stray too far from what would be seen in practice, and the results would have little meaning.) The results show that the DBA's feedback can still improve the quality of the recommendations significantly, despite the fact that WFIT has very inaccurate internal statistics.

Delayed Feedback. The previous experiments assumed that the DBA accepts the recommendation of WFIT after each query. In contrast, the next experiments evaluate the effect of delayed feedback, which is what we expect to see in practice. We model this scenario with a feedback input V_T, where the DBA requests and accepts the current recommendation of WFIT every T queries. This feedback renews the "lease" of the current recommendation, which in turn delays WFIT from switching to a potentially better recommendation. Hence, some degradation in performance is possible.

The results of this experiment are shown in Figure 11. The first curve shows the performance for $T = 1$, which grants full autonomy to WFIT. The other curves show the result of increasing the delay T to 25, 50, and 75. There is clearly a loss in overall performance when the responses of the DBA are delayed. At the end of the workload, the performance with $T > 1$ is around 85% of optimal, which is below the 95% level achieved by WFIT without the lag. A close examination of the results reveals that most indices are beneficial only for short windows of the workload, due to intervening updates that make indices expensive to maintain. This aspect of the workload makes the delayed responses particularly detrimental, and reflects our choice of this workload as a "stress test" for WFIT. However, it is important to observe that the performance does not continue to degrade as the length of the lag increases. We limited the lag to 75 queries in order to avoid a lag that spanned a large portion of the phase length of 200 queries. In general, the results suggest that semi-automatic interface can provide robust recommendations even when the lag is significant compared to the phase length.

Automatic Maintenance of Stable Partition. The final set of experiments examines the performance of WFIT when $chooseCands$ is used to maintain the stable partition automatically, as described in Section 5.2. In this case, the stable partition may change over time, which causes $repartition$ to be invoked. We compare this approach to the variation of WFIT with a fixed stable partition.

Figure 12 shows the performance of WFIT with a fixed stable partition and with automatic maintenance of candidates, labeled FIXED and AUTO, respectively. We see an overall improvement in the performance using $chooseCands$ to maintain the indices and interactions on-the-fly. Overall, *chooseCands* mined about 300 candidate indices from the workload, and changed the stable partition 147 times over the course of the experiment (although many of the calls to *repartition* only made minor changes to the modeled interactions). The observed performance clearly validates the ability of *repartition* to update the internal state of WFIT in a meaningful way. We also observe that the performance slightly exceeds OPT in the earlier queries, which are mostly read-only statements. This is due to the fact that the automatic maintenance of the stable partition allows WFIT to specialize the choice of indices for each phase, whereas OPT is limited to one set of candidates for the workload.

7 Conclusions

We introduced the novel paradigm of semi-automatic index tuning, and its realization in the WFIT algorithm. WFIT leverages and extends principled methods from online computation. Experimental results validate its numerous advantages over existing techniques, and the feasibility of semi-automatic tuning in practice.

8 References

[1] S. Agrawal, S. Chaudhuri, and V.Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases. In *VLDB*, 2000.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD*, 2006.

[3] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[4] A. Borodin, N. Linial, and M. E. Saks. An optimal on-line algorithm for metrical task system. *J. ACM*, 39(4), 1992.

[5] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *ICDE*, 2007.

[6] N. Bruno and S. Chaudhuri. Constrained physical design tuning. *PVLDB*, 1(1), 2008.

[7] N. Bruno and S. Chaudhuri. Interactive physical design tuning. In *ICDE*, 2010.

[8] N. Bruno and R. V. Nehme. Configuration-parametric query optimization for physical design tuning. In *SIGMOD*, 2008.

[9] L. Hu, K. A. Ross, Y.-C. Chang, C. A. Lang, and D. Zhang. Queryscope: visualizing queries for repeatable database tuning. *Proc. VLDB Endow.*, 1:1488–1491, August 2008.

[10] T. Malik, X. Wang, D. Dash, A. Chaudhary, A. Ailamaki, and R. C. Burns. Adaptive physical design for curated archives. In *SSDBM*, 2009.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, 1993.

[12] S. Papadomanolakis, D. Dash, and A. Ailamaki. Efficient use of the query optimizer for automated physical design. In *VLDB*, 2007.

[13] K.-U. Sattler, M. Lühring, I. Geist, and E. Schallehn. Autonomous management of soft indexes. In *SMDB*, 2007.

[14] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *SMDB*, 2007.

[15] K. Schnaitter and N. Polyzotis. A Benchmark for Online Index Selection. In *SMDB*, 2009.

[16] K. Schnaitter, N. Polyzotis, and L. Getoor. Index interactions in physical design tuning: modeling, analysis, and applications. *Proc. VLDB Endow.*, 2(1), 2009.

[17] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

APPENDIX

A Competitive Analysis

This section provides proofs of our results on the competitive ratio of WFA (Theorem 4.1) and WFA$^+$ (Theorem 4.3). The proof starts with two technical lemmas that lead to a central result given in Theorem A.1. This theorem leads immediately to Theorem 4.1, and it is also used to prove Theorem 4.3 with a bit more machinery.

We begin with notation. Fix the workload Q of N statements and denote the i-th query as q_i. In the context of this analysis, the algorithms choose recommendations that are subsets of a fixed set of candidate indices $\mathcal{C}$. Prior to observing any queries, the materialized set of indices is some set $S_0 \subseteq \mathcal{C}$.

We introduce a formal expression for the transition cost $\delta(X, Y)$:

$$\delta(X,Y) = \sum_{a \in Y - X} \delta^+(a) + \sum_{a \in X - Y} \delta^-(a)$$

where $\delta^+(a)$ and $\delta^-(a)$ denote respectively the cost of creating and dropping index a. We use the shorthand notation

$$\text{WFA}_n = \mathit{totWork}(\text{WFA}, Q_n, \emptyset)$$

for the total work of WFA on a prefix Q_n of the workload. We define the shorthand WFA^+_n and OPT_n similarly.

For each query q_i we fix a set S^*_i that minimizes the cost of q_i:

$$S^*_i = \arg\min_{X \subseteq C} \mathit{cost}(q_i, X).$$

The sum of these values over a prefix of the workload is denoted

$$\text{BASE}_n = \textstyle\sum_{i=1}^n \mathit{cost}(q_i, S^*_i).$$

Our analysis makes frequent use of these quantities as a simple lower bound on the query processing cost that must be paid by any algorithm. The following result shows that the minimum query cost also bounds the amount that the work function for an individual state increases after each statement.

LEMMA A.1. $w_{i+1}(S) \geq w_i(S) + \mathit{cost}(q_{i+1}, S^*_{i+1})$ *for all* $i \geq 0$.

Proof: For the case $i = 0$, we apply the triangle inequality of δ:

$$\begin{aligned} w_1(S) &= \min_{X \subseteq C} \{w_0(X) + \mathit{cost}(q_1, X) + \delta(X, S)\} \\ &= \min_{X \subseteq C} \{\delta(S_0, X) + \mathit{cost}(q_1, X) + \delta(X, S)\} \\ &\geq \delta(S_0, S) + \min_{X \subseteq C} \mathit{cost}(q_1, X) \\ &= w_0(S) + \mathit{cost}(q_1, S^*_1). \end{aligned}$$

For the case $i \geq 1$, note that there exist two index configurations P_i, P_{i-1} that satisfy the following equations:

$$\begin{aligned} w_{i+1}(S) &= w_i(P_i) + \mathit{cost}(q_{i+1}, P_i) + \delta(P_i, S) \\ w_i(P_i) &= w_{i-1}(P_{i-1}) + \mathit{cost}(q_i, P_{i-1}) + \delta(P_{i-1}, P_i) \end{aligned}$$

In other words, the final steps in the path corresponding to $w_{i+1}(S)$ are $P_{i-1} \to P_i \to S$. The path corresponding to $w_i(S)$ may have a different configuration as a predecessor to S, but we can use the path that passes through P_{i-1} to bound the value:

$$\begin{aligned} w_i(S) &\leq w_{i-1}(P_{i-1}) + \mathit{cost}(q_i, P_{i-1}) + \delta(P_{i-1}, S) \\ &= w_i(P_i) - \delta(P_{i-1}, P_i) + \delta(P_{i-1}, S) \\ &\leq w_i(P_i) + \delta(P_i, S) \end{aligned}$$

The triangle inequality yields the third step. It follows that

$$w_{i+1}(S) \geq w_i(S) + \mathit{cost}(q_{i+1}, P_i) \geq w_i(S) + \mathit{cost}(q_{i+1}, S^*_{i+1})$$

as desired.

We next give a result that shows the transition cost of a cyclic sequence of configurations does not change if we reverse the cycle.

LEMMA A.2. *Consider the sequence of index configurations $S_0, S_1, \ldots, S_n, S_0 \subseteq \mathcal{C}$. The following identity holds:*

$$\sum_{i=1}^{n} \delta(S_{i-1}, S_i) + \delta(S_n, S_0) = \sum_{i=1}^{n} \delta(S_i, S_{i-1}) + \delta(S_0, S_n)$$

Proof: By induction on n. The base cases $n = 0, 1$ are trivial, so we consider the interesting base case $n = 2$. For the cycle S_0, S_1, S_2, S_0, the total transition cost is equal to the sum of $\delta^+(a) + \delta^-(a)$ over all indices a that occur in exactly one or two of the sets S_0, S_1, S_2. This can be checked by enumerating the possible sets that such indices can occur in. By symmetry, the transitions on the reverse cycle S_0, S_2, S_1, S_0 have the same cost.

Now consider the inductive case $n \geq 3$. By an application of the inductive hypothesis to $S_0, S_1, \ldots, S_{n-1}, S_0$,

$$\begin{aligned} \sum_{i=1}^{n} \delta(S_{i-1}, S_i) + \delta(S_n, S_0) \quad = \quad & \sum_{i=1}^{n-1} \delta(S_i, S_{i-1}) - \delta(S_{n-1}, S_0) \\ & + \delta(S_0, S_{n-1}) + \delta(S_{n-1}, S_n) \\ & + \delta(S_n, S_0) \end{aligned}$$

The r.h.s. contains the cost of the cycle S_0, S_{n-1}, S_n, S_0. Since we assumed $n \geq 3$, we may apply the inductive hypothesis once more, and replace these terms with the cost of the reverse cycle. The lemma follows from this substitution. ▮

We can now prove the central result of this section. The theorem shows a bound on the cost of WFA that is strictly stronger than the competitive ratio of Theorem 4.1 when the minimum query costs $\mathit{cost}(q_i, S^*_i)$ are significant compared to the cost of the optimal schedule. The stronger statement is needed to prove Theorem 4.3.

THEOREM A.1. *The total work of* WFA *satisfies*

$$\textsc{wfa}_N - \textsc{base}_N \leq (2^{|\mathcal{C}|+1} - 1)(\textsc{opt}_N - \textsc{base}_N) + \alpha$$

where α does not depend on the workload Q.

Proof: We follow the overall strategy of the proof of Lemma 9.3 in [3], which is an analogous result for task systems with symmetric transition costs. Our setting differs since δ is not symmetric. The original proof also does not consider the effect of the terms $\mathit{cost}(q_i, S^*_i)$ that we account for in the theorem.

We start with notation borrowed from [3]:

$$\begin{aligned} \mu &= \max\{\delta(X, Y) \mid X, Y \subseteq \mathcal{C}\} \\ S_i &= \text{configuration recommended by } \textsc{wfa} \text{ for } q_i \\ B_i &= \textstyle\sum_S w_i(S) + \sum_{S \neq S_i} w_i(S) \end{aligned}$$

Our eventual goal is to derive separate bounds for $\textsc{wfa}_N$ and $\textsc{opt}_N$ with respect to B_N, and then combine these bounds. We first observe that

$$\begin{aligned} B_{i+1} - B_i &= w_{i+1}(S_i) - w_i(S_{i+1}) + 2 \cdot \sum_{S \neq S_i, S_{i+1}} (w_{i+1}(S) - w_i(S)) \\ &\quad + w_{i+1}(S_{i+1}) - w_i(S_{i+1}) + w_{i+1}(S_i) - w_i(S_i) \\ &\geq w_{i+1}(S_i) - w_i(S_{i+1}) \\ &\quad + (2^{|\mathcal{C}|+1} - 2) \cdot \mathit{cost}(q_{i+1}, S^*_{i+1}) \end{aligned}$$

by Lemma A.1. We can show that

$$w_{i+1}(S_i) - w_i(S_{i+1}) \geq \delta(S_{i+1}, S_i) + \mathit{cost}(q_{i+1}, S_{i+1})$$

using the same reasoning as [3] (see the original proof for details[4]). Hence,

$$\begin{aligned} B_{i+1} - B_i \quad \geq \quad & \delta(S_{i+1}, S_i) + \mathit{cost}(q_{i+1}, S_{i+1}) \\ & + (2^{|\mathcal{C}|+1} - 2) \cdot \mathit{cost}(q_{i+1}, S^*_{i+1}). \end{aligned}$$

Summing these inequalities for $0 \leq i \leq N - 1$ yields

$$\begin{aligned} B_N - B_0 \quad \geq \quad & \sum_{i=1}^{N} \delta(S_i, S_{i-1}) + \mathit{cost}(q_i, S_i) \\ & + (2^{|\mathcal{C}|+1} - 2) \cdot \textsc{base}_N. \end{aligned}$$

The first line of the r.h.s. is similar to the total work of WFA except that the transition costs are reversed. We can remedy this via Lemma A.2, which leads to

$$\begin{aligned} B_N - B_0 \quad \geq \quad & \sum_{i=1}^{N} \delta(S_{i-1}, S_i) + \mathit{cost}(q_i, S_i) \\ & + \delta(S_N, S_0) - \delta(S_0, S_N) \\ & + (2^{|\mathcal{C}|+1} - 2) \cdot \textsc{base}_N \\ = \quad & \textsc{wfa}_N - (2^{|\mathcal{C}|+1} - 2) \cdot \textsc{base}_N \\ & + \delta(S_N, S_0) - \delta(S_0, S_N) \end{aligned}$$

Finally, we can bound $\textsc{wfa}_N - \textsc{base}_N$ by

$$\begin{aligned} \textsc{wfa}_N - \textsc{base}_N \quad \leq \quad & B_N + (2^{|\mathcal{C}|+1} - 1) \cdot \textsc{base}_N \\ & - B_0 - \delta(S_N, S_0) + \delta(S_0, S_N) \end{aligned}$$

To complete the proof, we note that

$$B_N \leq (2^{|\mathcal{C}|+1} - 1) \cdot \textsc{opt}_N + (2^{|\mathcal{C}|+1} - 2)\mu$$

as shown in [3]. ▮

Proof of Theorem 4.1. Rearranging the terms in Theorem A.1, we have

$$\textsc{wfa}_N \leq (2^{|\mathcal{C}|+1} - 1)\textsc{opt}_N - (2^{|\mathcal{C}|+1} - 2)\textsc{base}_N + \alpha$$

where α does not depend on the workload Q. Since $(2^{|\mathcal{C}|+1} - 2)$ and $\textsc{base}_N$ are nonnegative, Theorem 4.1 follows. ▮

Proof of Theorem 4.3. We now show the competitive ratio of $\textsc{wfa}^+$ using a fixed stable partition $\{C_1, \ldots, C_K\}$ of the candidate indices $\mathcal{C}$. Our strategy is to use Theorem A.1 to analyze the recommendations chosen by WFA within each part C_k.

We first extend some of our previous notation to describe the behavior of WFA in an individual part C_k. We use $\textsc{wfa}^{(k)}$ to represent an instance of WFA that selects recommendations from the part C_k only. Similarly, $\textsc{opt}^{(k)}$ is the idealized algorithm that chooses the optimal recommendations from C_k with advance knowledge of the workload Q. Our shorthand for total work extends naturally, e.g., $\textsc{wfa}^{(k)}_n$ denotes the total work of the path that implements the

[4] The inequality from [3] reverses the arguments to δ. In the original proof, the distinction is not important because the transitions are assumed to be symmetric. In order to prove the competitive ratio in our setting, the arguments to δ are reversed in the criteria that WFA uses to select the next recommendation. This results in the slightly different inequality.

recommendations of $\textsc{wfa}^{(k)}$ for the first n queries. We denote

$$\begin{array}{rcl} S_i &=& \text{recommendation of } \textsc{wfa}^+ \text{ for } q_i \\ S_i^{(k)} \equiv S_i \cap C_k &=& \text{recommendation of } \textsc{wfa}^{(k)} \text{ for } q_i \\ O_i &=& \text{recommendation of } \textsc{opt} \text{ for } q_i \end{array}$$

It follows easily from (2.1) that $O_1 \cap C_k, \ldots, O_n \cap C_k$ is an optimal path within the part C_k of the stable partition. A similar fact is:

$$S_i^* \cap C_k = \arg \min_{X \subseteq C_k} \mathit{cost}(q_i, X)$$

In other words, the minimum query processing cost using configurations within C_k is achieved by $S_i^* \cap C_k$. Thus we use the notation

$$\textsc{base}_n^{(k)} = \textstyle\sum_{i=1}^n \mathit{cost}(q_i, S_i^* \cap C_k)$$

We can also break down the transition cost based on the partition:

$$\delta(X, Y) = \sum_k \delta(X \cap C_k, Y \cap C_k)$$

We need to express the total work of $\textsc{wfa}^+$ and $\textsc{opt}$ w.r.t. the lower bounds $\textsc{base}_N^{(k)}$ in order to apply Theorem A.1. Note that $\textsc{opt}$'s recommendations $O_0, \ldots, O_N$ obey the following identity:

$$\begin{array}{rl} & \mathit{cost}(q_i, O_i) - \mathit{cost}(q_i, S_i^*) \\ = & \mathit{benefit}_{q_i}(S_i^*, \emptyset) - \mathit{benefit}_{q_i}(O_i, \emptyset) \\ = & \displaystyle\sum_k \mathit{benefit}_{q_i}(S_i^* \cap C_k, \emptyset) - \mathit{benefit}_{q_i}(O_i \cap C_k, \emptyset) \\ = & \displaystyle\sum_k \mathit{cost}(q_i, O_i \cap C_k) - \mathit{cost}(q_i, S_i^* \cap C_k) \end{array}$$

Now we rewrite the total work of $\textsc{opt}$, offset by $\textsc{base}_N$:

$$\begin{array}{rl} & \textsc{opt}_N - \textsc{base}_N \\ = & \displaystyle\sum_{i=1}^N \delta(O_{i-1}, O_i) + \mathit{cost}(q_i, O_i) - \mathit{cost}(q_i, S_i^*) \\ = & \displaystyle\sum_i \delta(O_{i-1}, O_i) + \sum_k \mathit{cost}(q_i, O_i \cap C_k) - \mathit{cost}(q_i, S_i^* \cap C_k) \\ = & \displaystyle\sum_k \sum_i \delta(O_{i-1} \cap C_k, O_i \cap C_k) + \mathit{cost}(q_i, O_i \cap C_k) \\ & \qquad \displaystyle - \sum_i \mathit{cost}(q_i, S_i^* \cap C_k) \\ = & \displaystyle\sum_k \textsc{opt}_N^{(k)} - \textsc{base}_N^{(k)} \end{array}$$

Applying the same steps to the sequence $S_0, \ldots, S_N$, we can derive an analogous bound for the cost of $\textsc{wfa}^+$:

$$\textsc{wfa}_N^+ - \textsc{base}_N = \textstyle\sum_k \textsc{wfa}_N^{(k)} - \textsc{base}_N^{(k)}.$$

Now by Theorem A.1, we know that $\textsc{wfa}_N^{(k)} - \textsc{base}_N^{(k)}$ is bounded above by $(2^{|C_k|+1} - 1)(\textsc{opt}_N^{(k)} - \textsc{base}_N^{(k)}) + \alpha_k$ where α_k does not depend on the workload. Then we obviously have

$$\textsc{wfa}_N^{(k)} - \textsc{base}_N^{(k)} \le (2^{c_{\max}+1} - 1)(\textsc{opt}_N^{(k)} - \textsc{base}_N^{(k)}) + \alpha_k$$

where, as in the statement of Theorem 4.3, $c_{\max}$ is defined as the maximum of $|C_k|$ for all k. We apply this as follows:

$$\begin{array}{rl} \textsc{wfa}_N^+ = & \textsc{base}_N + \displaystyle\sum_k \textsc{wfa}_N^{(k)} - \textsc{base}_N^{(k)} \\ \le & \textsc{base}_N + (2^{c_{\max}+1} - 1) \displaystyle\sum_k (\textsc{opt}_N^{(k)} - \textsc{base}_N^{(k)}) + \sum_k \alpha_k \\ = & \textsc{base}_N + (2^{c_{\max}+1} - 1)(\textsc{opt}_N - \textsc{base}_N) + \displaystyle\sum_k \alpha_k \\ = & (2^{c_{\max}+1} - 1)\textsc{opt}_N - (2^{c_{\max}+1} - 2)\textsc{base}_N + \displaystyle\sum_k \alpha_k \\ \le & (2^{c_{\max}+1} - 1)\textsc{opt}_N + \displaystyle\sum_k \alpha_k \end{array}$$

This proves that the competitive ratio of $\textsc{wfa}^+$ is $2^{c_{\max}+1} - 1$. ■

B Proof of Theorem 4.2

Theorem 4.2 states that if $\textsc{wfa}^+$ uses any fixed stable partition $\{C_1, \ldots, C_K\}$ of the indices $\mathcal{C}$, it can generate the same recommendations as the naive application of $\textsc{wfa}$ that jointly tracks all subsets of $\mathcal{C}$. Before the main proof, we give one preliminary result that describes the relationship between the global work function and the work functions that $\textsc{wfa}^+$ maintains for each part.

LEMMA B.1. *Let w_n be the work function for the workload Q_n and the indices $\mathcal{C}$. Let $\mathbf{w}_n^{(k)}$ be the work function values calculated by $\textsc{wfa}^+$ for C_k after observing Q_n. For any $S \subseteq \mathcal{C}$,*

$$w_n(S) = \sum_k \mathbf{w}_n^{(k)}[S \cap C_k] - (K-1) \sum_{i=1}^n \mathit{cost}(q_i, \emptyset).$$

Proof: By induction on n. For the base case $n = 0$, the sum is empty, and the theorem follows from the identity

$$\delta(X, S) = \textstyle\sum_k \delta(X \cap C_k, S \cap C_k)$$

mentioned in earlier sections. Specifically, we have

$$w_0(S) = \delta(S_0, S) = \textstyle\sum_k \delta(S_0 \cap C_k, S \cap C_k) = \sum_k \mathbf{w}_0^{(k)}(S \cap C_k).$$

To prove the inductive case $n > 1$, we observe the following identity that follows easily from (2.1): for all $X \subseteq \mathcal{C}$,

$$\mathit{cost}(q_n, X) \;=\; \textstyle\sum_k \mathit{cost}(q_n, X \cap C_k) - (K-1)\mathit{cost}(q_n, \emptyset)$$

Recall the definition of the work function

$$w_n(S) = \min_{X \subseteq \mathcal{C}} \{w_{n-1}(X) + \mathit{cost}(q_n, X) + \delta(X, S)\}$$

If we apply the inductive hypothesis to $w_{n-1}(X)$ and also decompose the terms $\mathit{cost}(q_n, X)$ and $\delta(X, S)$ as shown above, the result simplifies to

$$\begin{array}{rcl} w_n(S) & = & \displaystyle\min_{X \subseteq \mathcal{C}} \{\textstyle\sum_k \mathbf{w}_{n-1}^{(k)}(X \cap C_k) \;+\; \mathit{cost}(q_n, X \cap C_k) \\ & & \qquad\qquad\qquad\qquad +\; \delta(X \cap C_k, S \cap C_k)\} \\ & & -\,(K-1) \textstyle\sum_{i=1}^n \mathit{cost}(q_i, \emptyset). \end{array}$$

The terms in the sum over k each depend on a disjoint part C_k, so the summation can be pulled out of the min operation to yield

$$\begin{array}{rl} w_n(S) = & \textstyle\sum_k \displaystyle\min_{X_k \subseteq C_k} \{\mathbf{w}_{n-1}^{(k)}(X_k) + \mathit{cost}(q_n, X_k) + \delta(X_k, S \cap C_k)\} \\ & -\,(K-1) \textstyle\sum_{i=1}^n \mathit{cost}(q_i, \emptyset) \\ = & \textstyle\sum_k \mathbf{w}_n^{(k)}[S \cap C_k] - (K-1) \sum_{i=1}^n \mathit{cost}(q_i, \emptyset). \quad ■ \end{array}$$

In order to prove the equivalence between $\textsc{wfa}^+$ and $\textsc{wfa}$, we must resolve the fact that the selection criteria of $\textsc{wfa}$ are not deterministic: if more than one configuration satisfies the criteria, the pseudocode does not specify which configuration is chosen. Thus, we assume a simple tie-breaker based on lexicographic ordering, as follows. Let $\{a_1, \ldots, a_{|\mathcal{C}|}\}$ denote the indices in $\mathcal{C}$. If $X, Y \subseteq \mathcal{C}$ and $X \neq Y$, consider the minimum value of d where X and Y differ on a_d, meaning that a_d is in the symmetric difference $X \ominus Y$. The lexicographic tie-breaking rule prefers X to Y iff $a_d \in X$.

We consider a workload Q_n of length n. Let S_0 be the initial configuration, $S_1, \ldots, S_n$ be the recommendations of WFA, and $S_1^k, \ldots, S_n^k$ be the recommendations of WFA$^+$ within each part C_k. The claim of Theorem 4.2 can be stated as $S_n = \bigcup_k S_n^k$ for $n \geq 0$. The proof proceeds by induction on n. The base case $n = 0$ is trivial, as both algorithms start with the same state.

Consider the inductive case $n \geq 1$. Assume for contradiction that $S_n \neq \bigcup_k S_n^k$. Take the minimum d where $a_d \in S_n \ominus \bigcup_k S_n^k$. Let C_ℓ be the part that contains a_d, implying either $a_d \in S_n^\ell - S_n$ or $a_d \in S_n - S_n^\ell$. We first consider the case $a_d \in S_n^\ell - S_n$. Let $\widehat{S}_n$ denote $(S_n - C_\ell) \cup S_n^\ell$ which is the result of modifying S_n to be consistent with WFA$^+$'s choice within C_ℓ. We immediately observe that the lexicographic tie-breaker prefers $\widehat{S}_n$ to S_n. The set $\widehat{S}_n$ also satisfies WFA's explicit tie-breaking constraint $\widehat{S}_n \in \mathbf{p}[\widehat{S}_n]$, by virtue of the fact that both S_n and S_n^ℓ satisfy the constraint. Hence, the only possible reason that WFA recommends S_n instead of $\widehat{S}_n$ must be that $\mathit{score}(S_n) < \mathit{score}(\widehat{S}_n)$, i.e.,

$$w_n(S_n) + \delta(S_n, S_{n-1}) < w_n(\widehat{S}_n) + \delta(\widehat{S}_n, S_{n-1}).$$

We may use Lemma B.1 to decompose both sides of the inequality according to the stable partition. Since S_n and $\widehat{S}_n$ agree on all indices outside of C_ℓ, we may cancel terms to yield:

$$\begin{aligned} &\mathbf{w}_n^{(k)}(S_n \cap C_\ell) + \delta(S_n \cap C_\ell, S_{n-1} \cap C_\ell) \\ &\qquad\qquad < \mathbf{w}_n^{(k)}(S_n^\ell) + \delta(S_n^\ell, S_{n-1} \cap C_\ell). \end{aligned}$$

I.e., $S_n \cap C_\ell$ has a better score than S_n^ℓ, contradicting the fact that WFA$^+$ recommends S_n^ℓ within C_ℓ.

The proof when $a_d \in S_n - S_n^\ell$ is completely symmetric: we contradict WFA's choice of S_n by showing $\widehat{S}_n$ has a lower score. ■